

C Pointers

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 4 byte.

Consider the following example to define a pointer which stores the address of an integer.

1. `int n = 10;`
2. `int* p = &n; // Variable p of type pointer is pointing to the address of the variable n of type integer.`

Declaring a pointer

The pointer in c language can be declared using * (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

1. `int *a; //pointer to int`
2. `char *c; //pointer to char`

Pointer Example

An example of using pointers to print the address and value is given below.

As you can see in the above figure, pointer variable stores the address of number variable, i.e., fff4. The value of number variable is 50. But the address of pointer variable p is aaa3.

By the help of * (**indirection operator**), we can print the value of pointer variable p.

Let's see the pointer example as explained for the above figure.

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

3. **int** number=50;
4. **int** *p;
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %x \n",p); // p contains the address of the number therefore printing p gives the address of number.
7. printf("Value of p variable is %d \n",*p); // As we know that * is used to dereference a pointer therefore if we print *p, we will get the value stored at the address contained by p.
8. **return** 0;
9. }

Output

```
Address of number variable is fff4
Address of p variable is fff4
Value of p variable is 50
```

Pointer to array

1. **int** arr[10];
2. **int** *p[10]=&arr; // Variable p of type pointer is pointing to the address of an integer array arr.

Pointer to a function

1. **void** show (**int**);
2. **void**(*p)(**int**) = &display; // Pointer p is pointing to the address of a function

Pointer to structure

1. **struct** st {
2. **int** i;
3. **float** f;
4. }ref;
5. **struct** st *p = &ref;

Advantage of pointer

- 1) Pointer **reduces the code** and **improves the performance**, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.
- 2) We can **return multiple values from a function** using the pointer.
- 3) It makes you able to **access any memory location** in the computer's memory.

Usage of pointer

There are many applications of pointers in c language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where the pointer is used.

2) Arrays, Functions, and Structures

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

Address Of (&) Operator

The address of operator '&' returns the address of a variable. But, we need to use %u to display the address of a variable.

1. `#include<stdio.h>`
2. `int main(){`
3. `int number=50;`
4. `printf("value of number is %d, address of number is %u",number,&number);`
5. `return 0;`
6. `}`

Output

```
value of number is 50, address of number is fff4
```

NULL Pointer

A pointer that is not assigned any value but NULL is known as the NULL pointer. If you don't have any address to be specified in the pointer at the time of declaration, you can assign NULL value. It will provide a better approach.

```
int *p=NULL;
```

In the most libraries, the value of the pointer is 0 (zero).

Pointer Program to swap two numbers without using the 3rd variable.

1. `#include<stdio.h>`
2. `int main(){`
3. `int a=10,b=20,*p1=&a,*p2=&b;`
- 4.
5. `printf("Before swap: *p1=%d *p2=%d",*p1,*p2);`
6. `*p1=*p1+*p2;`
7. `*p2=*p1-*p2;`
8. `*p1=*p1-*p2;`
9. `printf("\nAfter swap: *p1=%d *p2=%d",*p1,*p2);`
- 10.
11. `return 0;`
12. `}`

Output

```
Before swap: *p1=10 *p2=20
After swap: *p1=20 *p2=10
```

Reading complex pointers

There are several things which must be taken into the consideration while reading the complex pointers in C. Lets see the precedence and associativity of the operators which are used regarding pointers.

Operator	Precedence	Associativity
<code>()</code> , <code>[]</code>	1	Left to right

*, identifier	2	Right to left
Data type	3	-

Here, we must notice that,

- `()`: This operator is a bracket operator used to declare and define the function.
- `[]`: This operator is an array subscript operator
- `*`: This operator is a pointer operator.
- Identifier: It is the name of the pointer. The priority will always be assigned to this.
- Data type: Data type is the type of the variable to which the pointer is intended to point. It also includes the modifier like signed int, long, etc).

How to read the pointer: `int (*p)[10]`.

To read the pointer, we must see that `()` and `[]` have the equal precedence. Therefore, their associativity must be considered here. The associativity is left to right, so the priority goes to `()`.

Inside the bracket `()`, pointer operator `*` and pointer name (identifier) `p` have the same precedence. Therefore, their associativity must be considered here which is right to left, so the priority goes to `p`, and the second priority goes to `*`.

Assign the 3rd priority to `[]` since the data type has the last precedence. Therefore the pointer will look like following.

- `char` -> 4
- `*` -> 2
- `p` -> 1
- `[10]` -> 3

The pointer will be read as `p` is a pointer to an array of integers of size 10.

Example

How to read the following pointer?

1. `int (*p)(int (*)[2], int (*)void)`

Explanation

This pointer will be read as p is a pointer to such function which accepts the first parameter as the pointer to a one-dimensional array of integers of size two and the second parameter as the pointer to a function which parameter is void and return type is the integer.

C Double Pointer (Pointer to Pointer)

As we know that, a pointer is used to store the address of a variable in C. Pointer reduces the access time of a variable. However, In C, we can also define a pointer to store the address of another pointer. Such pointer is known as a double pointer (pointer to pointer). The first pointer is used to store the address of a variable whereas the second pointer is used to store the address of the first pointer. Let's understand it by the diagram given below.

The syntax of declaring a double pointer is given below.

1. `int **p; // pointer to a pointer which is pointing to an integer.`

Consider the following example.

1. `#include<stdio.h>`
2. `void main ()`
3. `{`
4. `int a = 10;`
5. `int *p;`
6. `int **pp;`
7. `p = &a; // pointer p is pointing to the address of a`
8. `pp = &p; // pointer pp is a double pointer pointing to the address of pointer p`
9. `printf("address of a: %x\n",p); // Address of a will be printed`
10. `printf("address of p: %x\n",pp); // Address of p will be printed`
11. `printf("value stored at p: %d\n",*p); // value stored at the address contained by p i.e. 10 will be printed`

12. `printf("value stored at pp: %d\n",**pp); // value stored at the address contained by the pointer
stoyred at pp`
13. `}`

Output

```
address of a: d26a8734  
address of p: d26a8738  
value stored at p: 10  
value stored at pp: 10
```

C double pointer example

Let's see an example where one pointer points to the address of another pointer.

As you can see in the above figure, p2 contains the address of p (fff2), and p contains the address of number variable (fff4).

1. `#include<stdio.h>`
2. `int main(){`
3. `int number=50;`
4. `int *p;//pointer to int`
5. `int **p2;//pointer to pointer`
6. `p=&number;//stores the address of number variable`
7. `p2=&p;`
8. `printf("Address of number variable is %x \n",&number);`
9. `printf("Address of p variable is %x \n",p);`
10. `printf("Value of *p variable is %d \n",*p);`
11. `printf("Address of p2 variable is %x \n",p2);`
12. `printf("Value of **p2 variable is %d \n",*p);`
13. `return 0;`
14. `}`

Output

```
Address of number variable is fff4  
Address of p variable is fff4  
Value of *p variable is 50  
Address of p2 variable is fff2  
Value of **p variable is 50
```

Pointer Arithmetic in C

We can perform arithmetic operations on the pointers like addition, subtraction, etc. However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer. In pointer-from-pointer subtraction, the result will be an integer value. Following arithmetic operations are possible on the pointer in C language:

- Increment
- Decrement
- Addition
- Subtraction
- Comparison

Incrementing Pointer in C

If we increment a pointer by 1, the pointer will start pointing to the immediate next location. This is somewhat different from the general arithmetic since the value of the pointer will get increased by the size of the data type to which the pointer is pointing.

We can traverse an array by using the increment operation on a pointer which will keep pointing to every element of the array, perform some operation on that, and update itself in a loop.

The Rule to increment the pointer is given below:

1. $\text{new_address} = \text{current_address} + i * \text{size_of}(\text{data type})$

Where i is the number by which the pointer get increased.

32-bit

For 32-bit int variable, it will be incremented by 4 bytes.

64-bit

For 64-bit int variable, it will be incremented by 4 bytes.

Let's see the example of incrementing pointer variable on 64-bit architecture.

1. `#include <stdio.h>`
2. `int main(){`
3. `int number=50;`
4. `int *p;//pointer to int`
5. `p=&number;//stores the address of number variable`
6. `printf("Address of p variable is %u \n",p);`
7. `p=p+1;`
8. `printf("After increment: Address of p variable is %u \n",p); // in our case, p will get incremented by 4 bytes.`
9. `return 0;`
10. `}`

Output

```
Address of p variable is 3214864300
After increment: Address of p variable is 3214864304
```

Traversing an array by using pointer

1. `#include <stdio.h>`
2. `void main ()`
3. `{`
4. `int arr[5] = {1, 2, 3, 4, 5};`
5. `int *p = arr;`
6. `int i;`
7. `printf("printing array elements...\n");`
8. `for(i = 0; i < 5; i++)`
9. `{`
10. `printf("%d ",*(p+i));`
11. `}`
12. `}`

Output

```
printing array elements...
1 2 3 4 5
```

Decrementing Pointer in C

Like increment, we can decrement a pointer variable. If we decrement a pointer, it will start pointing to the previous location. The formula of decrementing the pointer is given below:

1. $\text{new_address} = \text{current_address} - i * \text{size_of}(\text{data type})$

32-bit

For 32-bit int variable, it will be decremented by 2 bytes.

64-bit

For 64-bit int variable, it will be decremented by 4 bytes.

Let's see the example of decrementing pointer variable on 64-bit OS.

1. `#include <stdio.h>`
2. `void main(){`
3. `int number=50;`
4. `int *p;//pointer to int`
5. `p=&number;//stores the address of number variable`
6. `printf("Address of p variable is %u \n",p);`
7. `p=p-1;`
8. `printf("After decrement: Address of p variable is %u \n",p); // P will now point to the immediate previous location.`
9. `}`

Output

```
Address of p variable is 3214864300
After decrement: Address of p variable is 3214864296
```

C Pointer Addition

We can add a value to the pointer variable. The formula of adding value to pointer is given below:

1. $\text{new_address} = \text{current_address} + (\text{number} * \text{size_of}(\text{data type}))$

32-bit

For 32-bit int variable, it will add $2 * \text{number}$.

64-bit

For 64-bit int variable, it will add $4 * \text{number}$.

Let's see the example of adding value to pointer variable on 64-bit architecture.

1. `#include <stdio.h>`
2. `int main(){`
3. `int number=50;`
4. `int *p;//pointer to int`
5. `p=&number;//stores the address of number variable`
6. `printf("Address of p variable is %u \n",p);`
7. `p=p+3; //adding 3 to pointer variable`
8. `printf("After adding 3: Address of p variable is %u \n",p);`
9. `return 0;`
10. `}`

Output

```
Address of p variable is 3214864300
After adding 3: Address of p variable is 3214864312
```

As you can see, the address of p is 3214864300. But after adding 3 with p variable, it is 3214864312, i.e., $4*3=12$ increment. Since we are using 64-bit architecture, it increments 12. But if we were using 32-bit architecture, it was incrementing to 6 only, i.e., $2*3=6$. As integer value occupies 2-byte memory in 32-bit OS.

C Pointer Subtraction

Like pointer addition, we can subtract a value from the pointer variable. Subtracting any number from a pointer will give an address. The formula of subtracting value from the pointer variable is given below:

1. $\text{new_address} = \text{current_address} - (\text{number} * \text{size_of}(\text{data type}))$

32-bit

For 32-bit int variable, it will subtract $2 * \text{number}$.

64-bit

For 64-bit int variable, it will subtract $4 * \text{number}$.

Let's see the example of subtracting value from the pointer variable on 64-bit architecture.

1. `#include<stdio.h>`
2. `int main(){`
3. `int number=50;`
4. `int *p;//pointer to int`
5. `p=&number;//stores the address of number variable`
6. `printf("Address of p variable is %u \n",p);`
7. `p=p-3; //subtracting 3 from pointer variable`
8. `printf("After subtracting 3: Address of p variable is %u \n",p);`
9. `return 0;`
10. `}`

Output

```
Address of p variable is 3214864300
After subtracting 3: Address of p variable is 3214864288
```

You can see after subtracting 3 from the pointer variable, it is 12 ($4*3$) less than the previous address value.

However, instead of subtracting a number, we can also subtract an address from another address (pointer). This will result in a number. It will not be a simple arithmetic operation, but it will follow the following rule.

If two pointers are of the same type,

1. Address2 -

Address1 = (Subtraction of two addresses)/size of data type which pointer points

Consider the following example to subtract one pointer from another.

```
1. #include<stdio.h>
2. void main ()
3. {
4.     int i = 100;
5.     int *p = &i;
6.     int *temp;
7.     temp = p;
8.     p = p + 3;
9.     printf("Pointer Subtraction: %d - %d = %d",p, temp, p-temp);
10. }
```

Output

```
Pointer Subtraction: 1030585080 - 1030585068 = 3
```

Illegal arithmetic with pointers

There are various operations which can not be performed on pointers. Since, pointer stores address hence we must ignore the operations which may lead to an illegal address, for example, addition, and multiplication. A list of such operations is given below.

- Address + Address = illegal
- Address * Address = illegal
- Address % Address = illegal
- Address / Address = illegal
- Address & Address = illegal
- Address ^ Address = illegal
- Address | Address = illegal

- ~Address = illegal

Pointer to function in C

As we discussed in the previous chapter, a pointer can point to a function in C. However, the declaration of the pointer variable must be the same as the function. Consider the following example to make a pointer pointing to the function.

```
1. #include<stdio.h>
2. int addition ();
3. int main ()
4. {
5.     int result;
6.     int (*ptr)();
7.     ptr = &addition;
8.     result = (*ptr)();
9.     printf("The sum is %d",result);
10. }
11. int addition()
12. {
13.     int a, b;
14.     printf("Enter two numbers?");
15.     scanf("%d %d",&a,&b);
16.     return a+b;
17. }
```

Output

```
Enter two numbers?10 15
The sum is 25
```

Pointer to Array of functions in C

To understand the concept of an array of functions, we must understand the array of function. Basically, an array of the function is an array which contains the addresses of functions. In other words, the pointer to an array of functions is a pointer pointing to an array which contains the pointers to the functions. Consider the following example.

```
1. #include<stdio.h>
2. int show();
3. int showadd(int);
4. int (*arr[3])();
5. int (**ptr)[3]();
6.
7. int main ()
8. {
9.     int result1;
10.    arr[0] = show;
11.    arr[1] = showadd;
12.    ptr = &arr;
13.    result1 = (**ptr)();
14.    printf("printing the value returned by show : %d",result1);
15.    (**ptr+1)(result1);
16.}
17. int show()
18.{
19.    int a = 65;
20.    return a++;
21.}
22. int showadd(int b)
23.{
24.    printf("\nAdding 90 to the value returned by show: %d",b+90);
25.}
```

Output

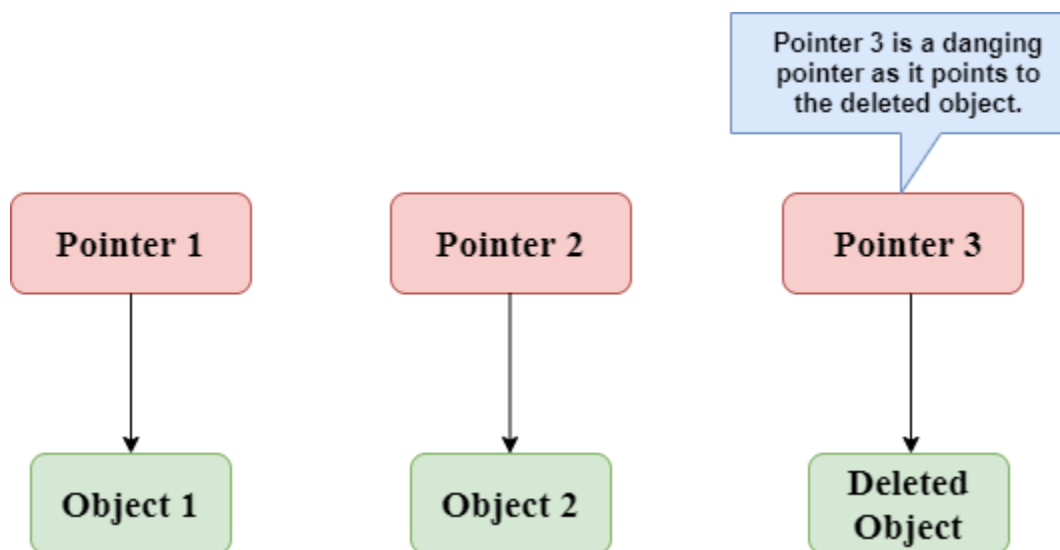
```
printing the value returned by show : 65
Adding 90 to the value returned by show: 155
```

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>

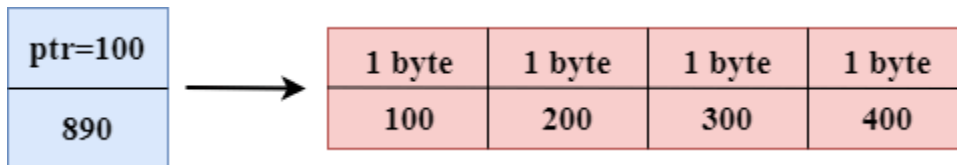

```

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. }

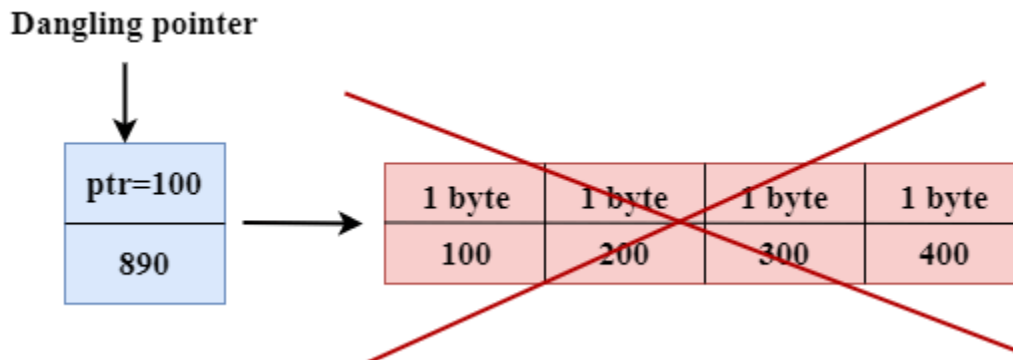
```

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.

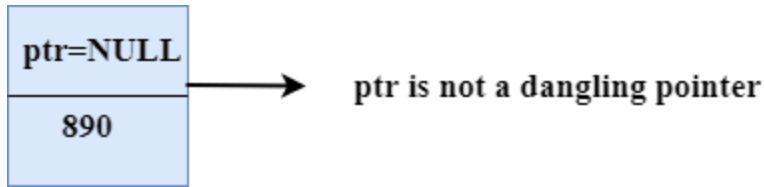
The statement **int *ptr=(int *)malloc(sizeof(int));** will allocate the memory with 4 bytes shown in the below image:



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.



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:



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.

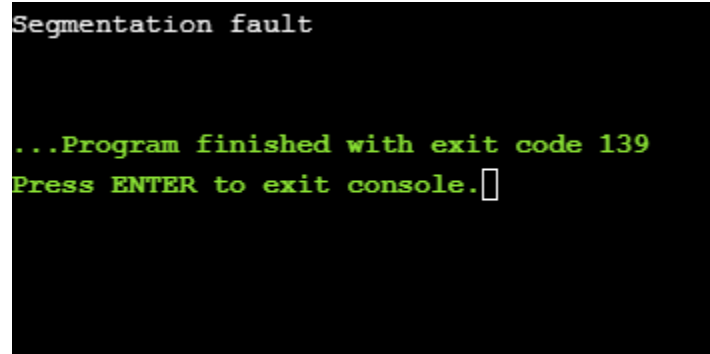
Let's understand through an example.

```
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.

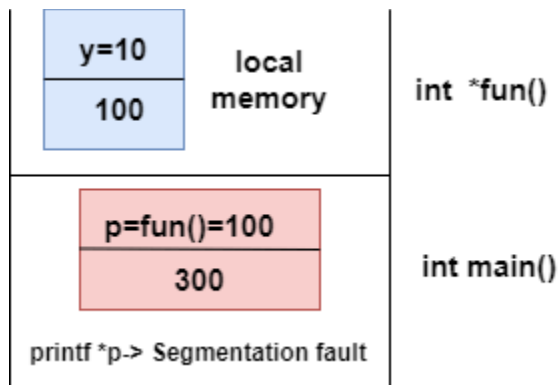
Output



```
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.

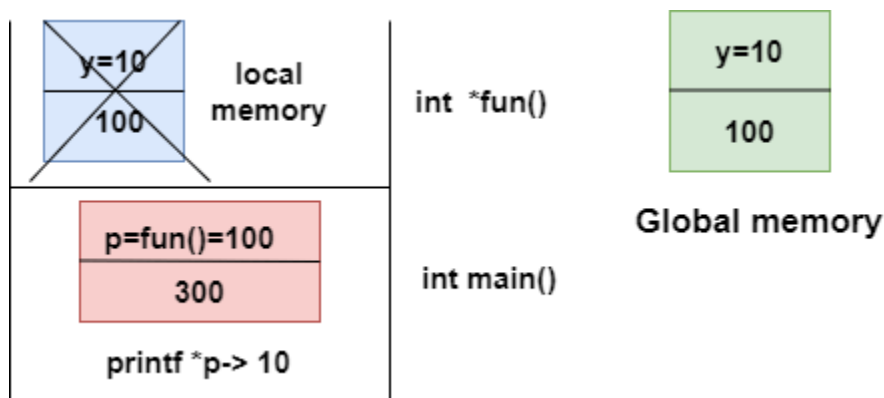
Output

```

10
...Program finished with exit code 0
Press ENTER to exit console.

```

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.

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. For example, we dynamically allocate the array space by using **sizeof()** operator:

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

In the above example, we use the `sizeof()` operator, which is applied to the cast of type `int`. We use **malloc()** function to allocate the memory and returns the pointer which is pointing to this allocated memory. The memory space is equal to the number of bytes occupied by the `int` data type and multiplied by 10.

Note:

The output can vary on different machines such as on 32-bit operating system will show different output, and the 64-bit operating system will show the different outputs of the same data types.

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.

1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `int x=89; // variable declaration.`
5. `printf("size of the variable x is %d", sizeof(x)); // Displaying the size of ?x? variable.`
6. `printf("\nsize of the integer data type is %d",sizeof(int)); //Displaying the size of integer data type.`

7. `printf("\nsize of the character data type is %d",sizeof(char)); //Displaying the size of character data type.`
- 8.
9. `printf("\nsize of the floating data type is %d",sizeof(float)); //Displaying the size of floating data type.`
10. `return 0;`
11. `}`

In the above code, we are printing the size of different data types such as int, char, float with the help of **sizeof()** operator.

Output

```
size of the variable x is 4
size of the integer data type is 4
size of the character data type is 1
size of the floating data type is 4

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

When operand is an expression

1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `double i=78.0; //variable initialization.`
5. `float j=6.78; //variable initialization.`
6. `printf("size of (i+j) expression is : %d",sizeof(i+j)); //Displaying the size of the expression (i+j).`
7. `return 0;`
8. `}`

In the above code, we have created two variables 'i' and 'j' of type double and float respectively, and then we print the size of the expression by using **sizeof(i+j)** operator.

Output

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

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:

- 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

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.

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;

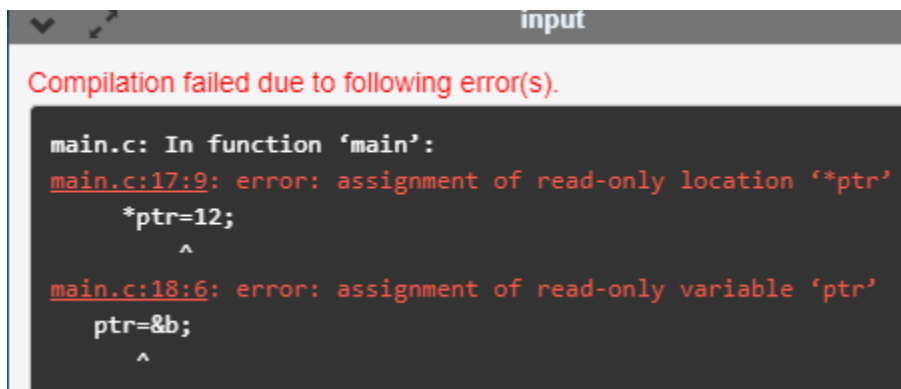
Let's understand through an example.

```
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.

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

1. **void** *pointer name;

Declaration of the void pointer is given below:

1. **void** *ptr;

In the above declaration, the void is the type of the pointer, and 'ptr' is the name of the pointer.

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:

```
1. #include <stdio.h>
2. int main()
3. {
4.     void *ptr = NULL; //void pointer
5.     int *p = NULL; // integer pointer
6.     char *cp = NULL; //character pointer
7.     float *fp = NULL; //float pointer
8.     //size of void pointer
9.     printf("size of void pointer = %d\n\n", sizeof(ptr));
10.    //size of integer pointer
11.    printf("size of integer pointer = %d\n\n", sizeof(p));
12.    //size of character pointer
13.    printf("size of character pointer = %d\n\n", sizeof(cp));
14.    //size of float pointer
15.    printf("size of float pointer = %d\n\n", sizeof(fp));
16.    return 0;
17. }
```

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.

1. `#include <stdio.h>`
2. `#include <malloc.h>`
3. `int main()`
4. `{`
5. `int a=90;`
6.
7. `int *x = (int*)malloc(sizeof(int));`
8. `x=&a;`
9. `printf("Value which is pointed by x pointer : %d",*x);`
10. `return 0;`
11. `}`

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:

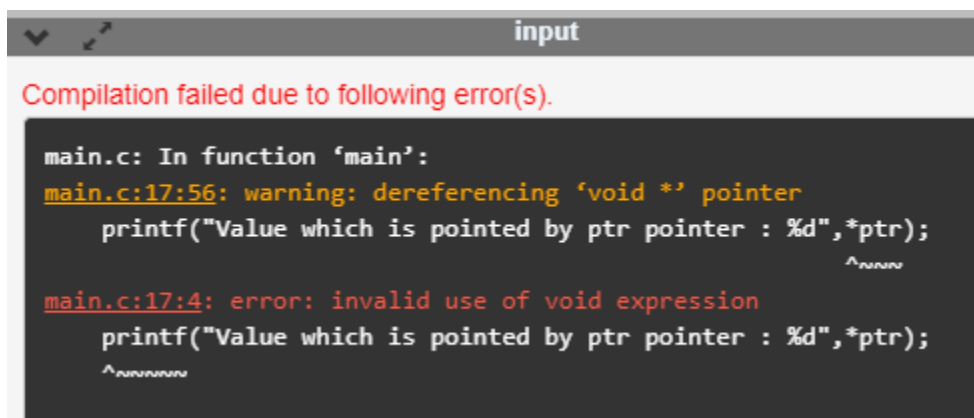
- **Dereferencing a void pointer in C**

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

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a=90;
5.     void *ptr;
6.     ptr=&a;
7.     printf("Value which is pointed by ptr pointer : %d",*ptr);
8.     return 0;
9. }
```

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.

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a=90;
5.     void *ptr;
6.     ptr=&a;
7.     printf("Value which is pointed by ptr pointer : %d",*(int*)ptr);
8.     return 0;
9. }
```

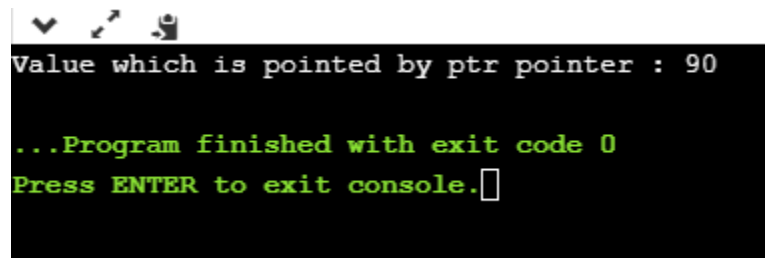
In the above code, we typecast the void pointer to the integer pointer by using the statement given below:

```
(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;
```

Output



```
Value which is pointed by ptr pointer : 90
...Program finished with exit code 0
Press ENTER to exit console.□
```

- **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:

```

1. #include<stdio.h>
2. int main()
3. {
4.     float a[4]={6.1,2.3,7.8,9.0};
5.     void *ptr;
6.     ptr=a;
7.     for(int i=0;i<4;i++)
8.     {
9.         printf("%f",*ptr);
10.        ptr=ptr+1;    // Incorrect.
11.
12.    }}

```

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.

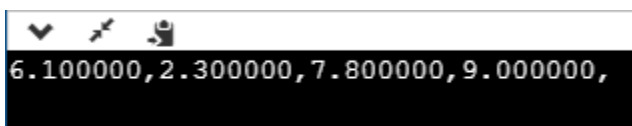
```

1. #include<stdio.h>
2. int main()
3. {
4.     float a[4]={6.1,2.3,7.8,9.0};
5.     void *ptr;
6.     ptr=a;
7.     for(int i=0;i<4;i++)
8.     {
9.         printf("%f",*((float*)ptr+i));
10.    }}

```

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,

```

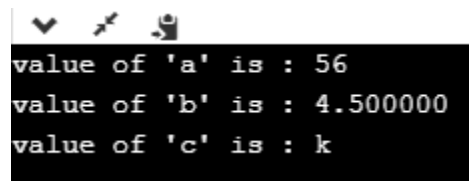
Why we use void pointers?

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.

```
1. #include<stdio.h>
2. int main()
3. {
4.     int a=56; // initialization of a integer variable 'a'.
5.     float b=4.5; // initialization of a float variable 'b'.
6.     char c='k'; // initialization of a char variable 'c'.
7.     void *ptr; // declaration of void pointer.
8.     // assigning the address of variable 'a'.
9.     ptr=&a;
10.    printf("value of 'a' is : %d",*((int*)ptr));
11.    // assigning the address of variable 'b'.
12.    ptr=&b;
13.    printf("\nvalue of 'b' is : %f",*((float*)ptr));
14.    // assigning the address of variable 'c'.
15.    ptr=&c;
16.    printf("\nvalue of 'c' is : %c",*((char*)ptr));
17.    return 0;
18.}
```

Output



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

C dereference pointer

As we already know that "**what is a pointer**", a pointer is a variable that stores the address of another variable. The dereference operator is also known as an indirection operator, which is represented by (*). When indirection operator (*) is used with the pointer variable, then it is known as **dereferencing a pointer**. When we dereference a pointer, then the value of the variable pointed by this pointer will be returned.

Why we use dereferencing pointer?

Dereference a pointer is used because of the following reasons:

- It can be used to access or manipulate the data stored at the memory location, which is pointed by the pointer.
- Any operation applied to the dereferenced pointer will directly affect the value of the variable that it points to.

Let's observe the following steps to dereference a pointer.

- First, we declare the integer variable to which the pointer points.

1. `int x =9;`

- Now, we declare the integer pointer variable.

1. `int *ptr;`

- After the declaration of an integer pointer variable, we store the address of 'x' variable to the pointer variable 'ptr'.

1. `ptr=&x;`

- We can change the value of 'x' variable by dereferencing a pointer 'ptr' as given below:

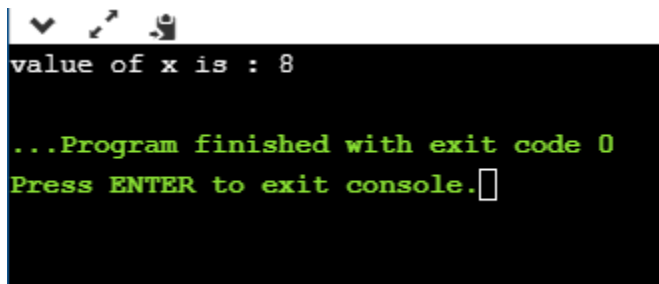
1. `*ptr =8;`

The above line changes the value of 'x' variable from 9 to 8 because 'ptr' points to the 'x' location and dereferencing of 'ptr', i.e., *ptr=8 will update the value of x.

Let's combine all the above steps:

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** x=9;
5. **int** *ptr;
6. ptr=&x;
7. *ptr=8;
8. printf("value of x is : %d", x);
9. **return** 0;}

Output



```
value of x is : 8

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

Let's consider another example.

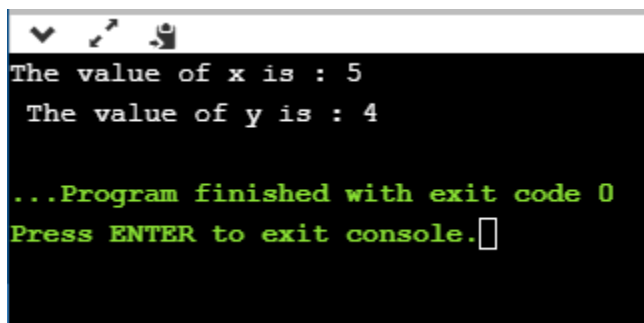
1. #include <stdio.h>
2. **int** main()
3. {
4. **int** x=4;
5. **int** y;
6. **int** *ptr;
7. ptr=&x;
8. y=*ptr;
9. *ptr=5;
10. printf("The value of x is : %d",x);
11. printf("\n The value of y is : %d",y);
12. **return** 0;
13. }

In the above code:

- We declare two variables 'x' and 'y' where 'x' is holding a '4' value.
- We declare a pointer variable 'ptr'.
- After the declaration of a pointer variable, we assign the address of the 'x' variable to the pointer 'ptr'.
- As we know that the 'ptr' contains the address of 'x' variable, so '*ptr' is the same as 'x'.
- We assign the value of 'x' to 'y' with the help of 'ptr' variable, i.e., y=***ptr** instead of using the 'x' variable.

Note: According to us, if we change the value of 'x', then the value of 'y' will also get changed as the pointer 'ptr' holds the address of the 'x' variable. But this does not happen, as 'y' is storing the local copy of value '5'.

Output



```
The value of x is : 5
The value of y is : 4

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

Let's consider another scenario.

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** a=90;
5. **int** *ptr1,*ptr2;
6. ptr1=&a;
7. ptr2=&a;
8. *ptr1=7;
9. *ptr2=6;
10. printf("The value of a is : %d",a);
11. **return** 0;
12. }

In the above code:

- First, we declare an 'a' variable.
- Then we declare two pointers, i.e., ptr1 and ptr2.
- Both the pointers contain the address of 'a' variable.
- We assign the '7' value to the *ptr1 and '6' to the *ptr2. The final value of 'a' would be '6'.

Note: If we have more than one pointer pointing to the same location, then the change made by one pointer will be the same as another pointer.

Output

```
The value of a is : 6
```