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 2 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.

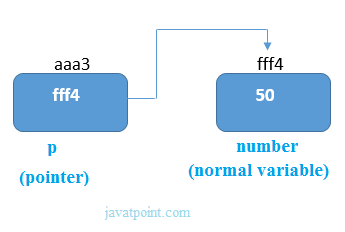
**Declaring a pointer**

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

# 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 stoted 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

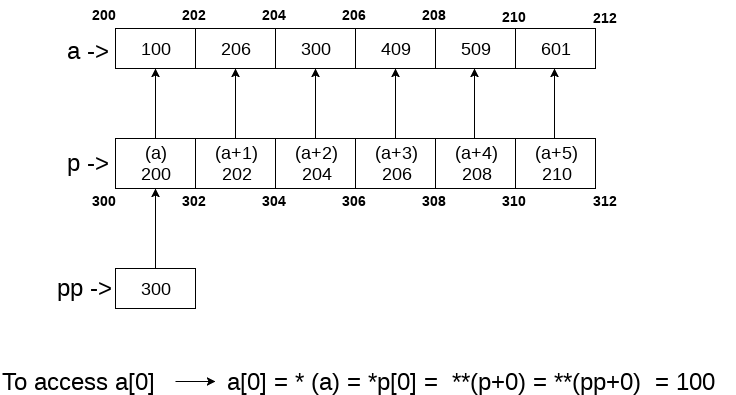
Address of p2 variable is fff2

Value of \*\*p variable is 50

## Q. What will be the output of the following program?

1. #include<stdio.h>
2. void main ()
3. {
4. int a[10] = {100, 206, 300, 409, 509, 601}; //Line 1
5. int \*p[] = {a, a+1, a+2, a+3, a+4, a+5}; //Line 2
6. int \*\*pp = p; //Line 3
7. pp++; // Line 4
8. printf("%d %d %d\n",pp-p,\*pp - a,\*\*pp); // Line 5
9. \*pp++; // Line 6
10. printf("%d %d %d\n",pp-p,\*pp - a,\*\*pp); // Line 7
11. ++\*pp; // Line 8
12. printf("%d %d %d\n",pp-p,\*pp - a,\*\*pp); // Line 9
13. ++\*\*pp; // Line 10
14. printf("%d %d %d\n",pp-p,\*pp - a,\*\*pp); // Line 11
15. }

#### Explanation



In the above question, the pointer arithmetic is used with the double pointer. An array of 6 elements is defined which is pointed by an array of pointer p. The pointer array p is pointed by a double pointer pp. However, the above image gives you a brief idea about how the memory is being allocated to the array a and the pointer array p. The elements of p are the pointers that are pointing to every element of the array a. Since we know that the array name contains the base address of the array hence, it will work as a pointer and can the value can be traversed by using \*(a), \*(a+1), etc. As shown in the image, a[0] can be accessed in the following ways.

* a[0]: it is the simplest way to access the first element of the array
* \*(a): since a store the address of the first element of the array, we can access its value by using indirection pointer on it.
* \*p[0]: if a[0] is to be accessed by using a pointer p to it, then we can use indirection operator (\*) on the first element of the pointer array p, i.e., \*p[0].
* \*\*(pp): as pp stores the base address of the pointer array, \*pp will give the value of the first element of the pointer array that is the address of the first element of the integer array. \*\*p will give the actual value of the first element of the integer array.

Coming to the program, Line 1 and 2 declare the integer and pointer array relatively. Line 3 initializes the double pointer to the pointer array p. As shown in the image, if the address of the array starts from 200 and the size of the integer is 2, then the pointer array will contain the values as 200, 202, 204, 206, 208, 210. Let us consider that the base address of the pointer array is 300; the double pointer pp contains the address of pointer array, i.e., 300. Line number 4 increases the value of pp by 1, i.e., pp will now point to address 302.

Line number 5 contains an expression which prints three values, i.e., pp - p, \*pp - a, \*\*pp. Let's calculate them each one of them.

* pp = 302, p = 300 => pp-p = (302-300)/2 => pp-p = 1, i.e., 1 will be printed.
* pp = 302, \*pp = 202, a = 200 => \*pp - a = 202 - 200 = 2/2 = 1, i.e., 1 will be printed.
* pp = 302, \*pp = 202, \*(\*pp) = 206, i.e., 206 will be printed.

Therefore as the result of line 5, The output 1, 1, 206 will be printed on the console. On line 6, \*pp++ is written. Here, we must notice that two unary operators \* and ++ will have the same precedence. Therefore, by the rule of associativity, it will be evaluated from right to left. Therefore the expression \*pp++ can be rewritten as (\*(pp++)). Since, pp = 302 which will now become, 304. \*pp will give 204.

On line 7, again the expression is written which prints three values, i.e., pp-p, \*pp-a, \*pp. Let's calculate each one of them.

* pp = 304, p = 300 => pp - p = (304 - 300)/2 => pp-p = 2, i.e., 2 will be printed.
* pp = 304, \*pp = 204, a = 200 => \*pp-a = (204 - 200)/2 = 2, i.e., 2 will be printed.
* pp = 304, \*pp = 204, \*(\*pp) = 300, i.e., 300 will be printed.

Therefore, as the result of line 7, The output 2, 2, 300 will be printed on the console. On line 8, ++\*pp is written. According to the rule of associativity, this can be rewritten as, (++(\*(pp))). Since, pp = 304, \*pp = 204, the value of \*pp = \*(p[2]) = 206 which will now point to a[3].

On line 9, again the expression is written which prints three values, i.e., pp-p, \*pp-a, \*pp. Let's calculate each one of them.

* pp = 304, p = 300 => pp - p = (304 - 300)/2 => pp-p = 2, i.e., 2 will be printed.
* pp = 304, \*pp = 206, a = 200 => \*pp-a = (206 - 200)/2 = 3, i.e., 3 will be printed.
* pp = 304, \*pp = 206, \*(\*pp) = 409, i.e., 409 will be printed.

Therefore, as the result of line 9, the output 2, 3, 409 will be printed on the console. On line 10, ++\*\*pp is writen. according to the rule of associativity, this can be rewritten as, (++(\*(\*(pp)))). pp = 304, \*pp = 206, \*\*pp = 409, ++\*\*pp => \*pp = \*pp + 1 = 410. In other words, a[3] = 410.

On line 11, again the expression is written which prints three values, i.e., pp-p, \*pp-a, \*pp. Let's calculate each one of them.

* pp = 304, p = 300 => pp - p = (304 - 300)/2 => pp-p = 2, i.e., 2 will be printed.
* pp = 304, \*pp = 206, a = 200 => \*pp-a = (206 - 200)/2 = 3, i.e., 3 will be printed.
* On line 8, \*\*pp = 410.

Therefore as the result of line 9, the output 2, 3, 410 will be printed on the console.

At last, the output of the complete program will be given as:

**Output**

1 1 206

2 2 300

2 3 409

2 3 410

# 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. new\_address= current\_address + i \* size\_of(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 2 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. new\_address= current\_address - i \* size\_of(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 immidiate 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. new\_address= current\_address + (number \* size\_of(data type))

### 32-bit

For 32-bit int variable, it will add 2 \* number.

### 64-bit

For 64-bit int variable, it will add 4 \* 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. new\_address= current\_address - (number \* size\_of(data type))

### 32-bit

For 32-bit int variable, it will subtract 2 \* number.

### 64-bit

For 64-bit int variable, it will subtract 4 \* 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

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 an 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])();
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

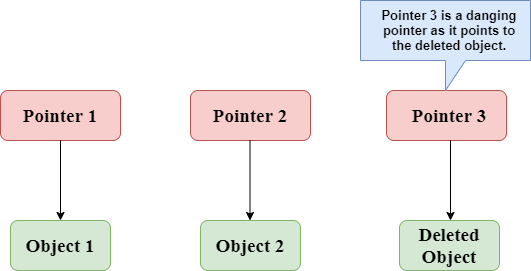
Adding 90 to the value returned by show: 155

# 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.

# 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.

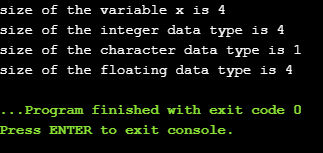
* **Operand is a data type**
* **Operand is an expression**

### 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.
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**



### 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;

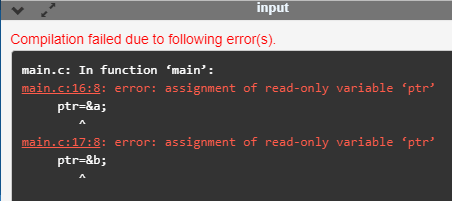
**Let's understand the constant pointer through an example.**

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.

# 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

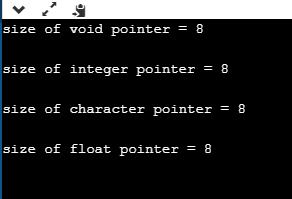
### Size of the void pointer in C

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**



### 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;
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**

void pointer in C

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

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

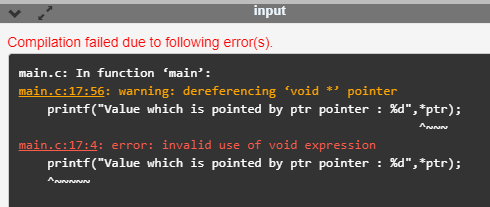
* **Dereferencing a void pointer in C**

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**



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. }

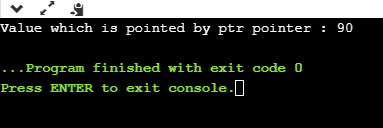
In the above code, we typecast the void pointer to the integer pointer by using the statement given below:

**(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**



* **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.
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**

void pointer in C

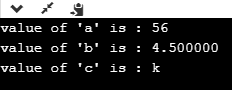
### Why we use void pointers?

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**



# What is a Null Pointer?

A Null Pointer is a pointer that does not point to any memory location. It stores the base address of the segment. The null pointer basically stores the Null value while void is the type of the pointer.

A null pointer is a special reserved value which is defined in a **stddef** header file. Here, Null means that the pointer is referring to the 0th memory location.

If we do not have any address which is to be assigned to the pointer, then it is known as a null pointer. When a NULL value is assigned to the pointer, then it is considered as a **Null pointer**.

## Applications of Null Pointer

**Following are the applications of a Null pointer:**

* It is used to initialize o pointer variable when the pointer does not point to a valid memory address.
* It is used to perform error handling with pointers before dereferencing the pointers.
* It is passed as a function argument and to return from a function when we do not want to pass the actual memory address.

## Examples of Null Pointer

int \*ptr=(int \*)0;  
float \*ptr=(float \*)0;  
char \*ptr=(char \*)0;  
double \*ptr=(double \*)0;  
char \*ptr='\0';  
int \*ptr=NULL;

**Let's look at the situations where we need to use the null pointer.**

* **When we do not assign any memory address to the pointer variable.**

1. #include <stdio.h>
2. int main()
3. {
4. int \*ptr;
5. printf("Address: %d", ptr); // printing the value of ptr.
6. printf("Value: %d", \*ptr); // dereferencing the illegal pointer
7. return 0;
8. }

In the above code, we declare the pointer variable \*ptr, but it does not contain the address of any variable. The dereferencing of the uninitialized pointer variable will show the compile-time error as it does not point any variable. According to the stack memory concept, the local variables of a function are stored in the stack, and if the variable does not contain any value, then it shows the garbage value. The above program shows some unpredictable results and causes the program to crash. Therefore, we can say that keeping an uninitialized pointer in a program can cause serious harm to the computer.

**How to avoid the above situation?**

We can avoid the above situation by using the Null pointer. A null pointer is a pointer pointing to the 0th memory location, which is a reserved memory and cannot be dereferenced.

1. #include <stdio.h>
2. int main()
3. {
4. int \*ptr=NULL;
5. if(ptr!=NULL)
6. {
7. printf("value of ptr is : %d",\*ptr);
8. }
9. else
10. {
11. printf("Invalid pointer");
12. }
13. return 0;
14. }

In the above code, we create a pointer **\*ptr** and assigns a **NULL** value to the pointer, which means that it does not point any variable. After creating a pointer variable, we add the condition in which we check whether the value of a pointer is null or not.

# C Function Pointer

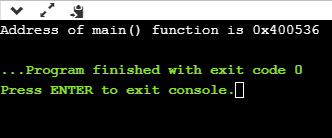
As we know that we can create a pointer of any data type such as int, char, float, we can also create a pointer pointing to a function. The code of a function always resides in memory, which means that the function has some address. We can get the address of memory by using the function pointer.

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**



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