

C++ Pointers

The pointer in C++ language is a variable, it is also known as locator or indicator that points to an address of a value.

The symbol of an address is represented by a pointer. In addition to creating and modifying dynamic data structures, they allow programs to emulate call-by-reference. One of the principal applications of pointers is iterating through the components of arrays or other data structures. The pointer variable that refers to the same data type as the variable you're dealing with has the address of that variable set to it (such as an int or string).

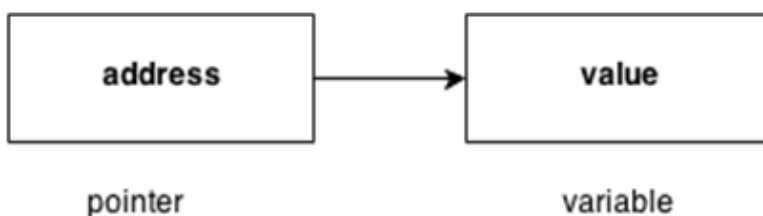
Syntax

```
datatype *var_name;  
int *ptr; // ptr can point to an address which holds int data
```

How to use a pointer?

1. Establish a pointer variable.
2. employing the unary operator (&), which yields the address of the variable, to assign a pointer to a variable's address.
3. Using the unary operator (*), which gives the variable's value at the address provided by its argument, one can access the value stored in an address.

Since the data type knows how many bytes the information is held in, we associate it with a reference. The size of the data type to which a pointer points is added when we increment a pointer.



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 function using pointer.

3) It makes you able to access any memory location in the computer's memory.

Usage of pointer

There are many usage of pointers in C++ language.

1) Dynamic memory allocation

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

Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the address of a variable.
* (asterisk sign)	Indirection operator	Access the value of an address.

Declaring a pointer

The pointer in C++ language can be declared using * (asterisk symbol).

```
int * a; //pointer to int
char * c; //pointer to char
```

AD

Pointer Example

Let's see the simple example of using pointers printing the address and value.

```
#include <iostream>
using namespace std;
```

```
int main()
{
    int number=30;
    int * p;
    p=&number;//stores the address of number variable
    cout<<"Address of number variable is:"<<&number<<endl;
    cout<<"Address of p variable is:"<<p<<endl;
    cout<<"Value of p variable is:"<<*p<<endl;
    return 0;
}
```

Output:

```
Address of number variable is:0x7ffccc8724c4
Address of p variable is:0x7ffccc8724c4
Value of p variable is:30
```

Pointer Program to swap 2 numbers without using 3rd variable

```
#include <iostream>
using namespace std;
int main()
{
    int a=20,b=10,*p1=&a,*p2=&b;
    cout<<"Before swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
    *p1=*p1+*p2;
    *p2=*p1-*p2;
    *p1=*p1-*p2;
    cout<<"After swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
    return 0;
}
```

Output:

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

What are Pointer and string literals?

String literals are arrays of character sequences with null ends. The elements of a string literal are arrays of type `const char` (because characters in a string cannot be modified) plus a terminating null-character.

What is a void pointer?

This unique type of pointer, which is available in C++, stands in for the lack of a kind. Pointers that point to a value that has no type are known as void pointers (and thus also an undetermined length and undetermined dereferencing properties). This indicates that void pointers are very flexible because they can point to any data type. This flexibility has benefits. Direct dereference is not possible with these pointers. Before they may be dereferenced, they must be converted into another pointer type that points to a specific data type.

AD

What is a invalid pointer?

A pointer must point to a valid address, not necessarily to useful items (like for arrays). We refer to these as incorrect pointers. Additionally, incorrect pointers are uninitialized pointers.

```
int *ptr1;  
int arr[10];  
int *ptr2 = arr+20;
```

Here, `ptr1` is not initialized, making it invalid, and `ptr2` is outside of the bounds of `arr`, making it likewise weak. (Take note that not all build failures are caused by faulty references.)

What is a null pointer?

A null pointer is not merely an incorrect address; it also points nowhere. Here are two ways to mark a pointer as NULL:

```
int *ptr1 = 0;  
int *ptr2 = NULL;
```

What is a pointer to a pointer?

In C++, we have the ability to build a pointer to another pointer, which might then point to data or another pointer. The unary operator (*) is all that is needed in the syntax for declaring the pointer for each level of indirection.

```
char a;  
char *b;  
char **c;  
a = 'g';  
b = &a;  
c = &b;
```

Here b points to a char that stores 'g', and c points to the pointer b.

AD

What are references and pointers?

1. Call-By-Value
2. Call-By-Reference with a Pointer Argument
3. Call-By-Reference with a Reference Argument

Example

```
#include  
using namespace std;  
// Pass-by-Value  
int square1(int n)  
{cout << "address of n1 in square1(): " << &n << "\n";  
n *= n;  
return n;  
}  
// Pass-by-Reference with Pointer Arguments  
void square2(int* n)  
{
```

```
cout << "address of n2 in square2(): " << n << "\n";
*n *= *n;
}

// Pass-by-Reference with Reference Arguments
void square3(int& n)
{

cout << "address of n3 in square3(): " << &n << "\n";
n *= n;
}

void example()
{
    // Call-by-Value
    int n1 = 8;
    cout << "address of n1 in main(): " << &n1 << "\n";
    cout << "Square of n1: " << square1(n1) << "\n";
    cout << "No change in n1: " << n1 << "\n";

    // Call-by-Reference with Pointer Arguments
    int n2 = 8;
    cout << "address of n2 in main(): " << &n2 << "\n";
    square2(&n2);
    cout << "Square of n2: " << n2 << "\n";
    cout << "Change reflected in n2: " << n2 << "\n";

    // Call-by-Reference with Reference Arguments
    int n3 = 8;
    cout << "address of n3 in main(): " << &n3 << "\n";
    square3(n3);
    cout << "Square of n3: " << n3 << "\n";
    cout << "Change reflected in n3: " << n3 << "\n";
}

// Driver program
int main() { example(); }
```

Output

```
address of n1 in main(): 0x7fffa7e2de64
address of n1 in square1(): 0x7fffa7e2de4c
Square of n1: 64
No change in n1: 8
address of n2 in main(): 0x7fffa7e2de68
address of n2 in square2(): 0x7fffa7e2de68
Square of n2: 64
Change reflected in n2: 64
address of n3 in main(): 0x7fffa7e2de6c
address of n3 in square3(): 0x7fffa7e2de6c
Square of n3: 64
Change reflected in n3: 64
```

[< Prev](#)[Next >](#)

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

sizeof() operator in C++

The sizeof() is an operator that evaluates the size of data type, constants, variable. It is a compile-time operator as it returns the size of any variable or a constant at the compilation time.

The size, which is calculated by the sizeof() operator, is the amount of RAM occupied in the computer.

Syntax of the sizeof() operator is given below:

```
sizeof(data_type);
```

In the above syntax, the data_type can be the data type of the data, variables, constants, unions, structures, or any other user-defined data type.

The sizeof () operator can be applied to the following operand types:

- **When an operand is of data type**

If the parameter of a **sizeof()** operator contains the data type of a variable, then the **sizeof()** operator will return the size of the data type.

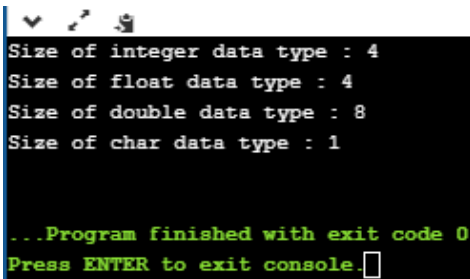
Let's understand this scenario through an example.

```
#include <iostream>
using namespace std;
int main()
{
    // Determining the space in bytes occupied by each data type.
    std::cout << "Size of integer data type : " << sizeof(int) << std::endl;
    std::cout << "Size of float data type : " << sizeof(float) << std::endl;
    std::cout << "Size of double data type : " << sizeof(double) << std::endl;
    std::cout << "Size of char data type : " << sizeof(char) << std::endl;
    return 0;
}
```

In the above program, we have evaluated the size of the in-built data types by using the sizeof() operator. As we know that int occupies 4 bytes, float occupies 4 bytes, double occupies 8 bytes, and char occupies 1 byte, and the same result is shown by the sizeof() operator as we can observe in the

following output.

Output



```
Size of integer data type : 4
Size of float data type : 4
Size of double data type : 8
Size of char data type : 1

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

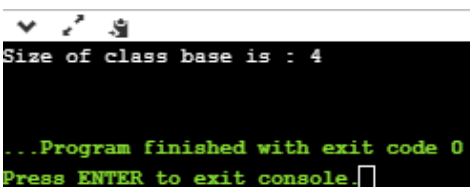
- When an operand is of Class type.

 **a new way to hire talent**

```
#include <iostream>
using namespace std;
class Base
{
    int a;
};
int main()
{
    Base b;
    std::cout << "Size of class base is : "<<sizeof(b) << std::endl;
    return 0;
}
```

In the above program, we have evaluated the size of the class, which is having a single integer variable. The output would be 4 bytes as int variable occupies 4 bytes.

Output



```
Size of class base is : 4

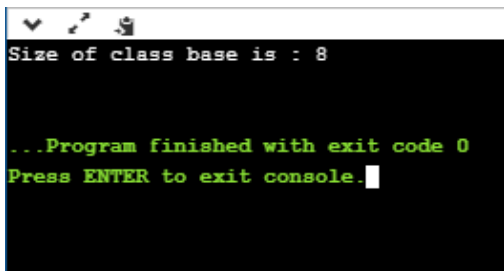
...Program finished with exit code 0
Press ENTER to exit console.
```

If we add one more integer variable in a class, then the code would look like:

```
#include <iostream>
using namespace std;
class Base
{
    int a;
    int d;
};
int main()
{
    Base b;
    std::cout << "Size of class base is : " << sizeof(b) << std::endl;
    return 0;
}
```

In the above code, we have added one more integer variable. In this case, the size of the class would be 8 bytes as **int** variable occupies 4 bytes, so two integer variables occupy 8 bytes.

Output



```
Size of class base is : 8

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

If we add a char variable in the above code, then the code would look like:

```
#include <iostream>

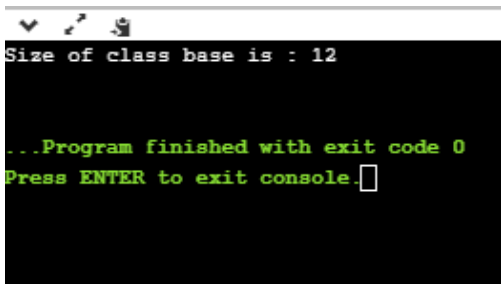
using namespace std;

class Base
{
    int a;
    int d;
    char ch;
```

```
};  
  
int main()  
{  
    Base b;  
    std::cout << "Size of class base is : " << sizeof(b) << std::endl;  
    return 0;  
}
```

In the above code, the class has two integer variables, and one char variable. According to our calculation, the size of the class would be equal to 9 bytes (int+int+char), but this is wrong due to the concept of structure padding.

Output



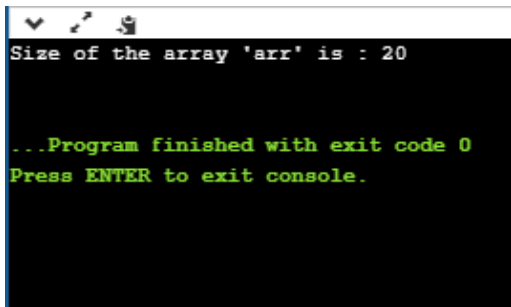
- **When an operand is of array type.**

```
#include <iostream>  
  
using namespace std;  
  
int main()  
{  
    int arr[]={10,20,30,40,50};  
    std::cout << "Size of the array 'arr' is : " << sizeof(arr) << std::endl;  
    return 0;  
}
```

In the above program, we have declared an array of integer type which contains five elements. We have evaluated the size of the array by using **sizeof()** operator. According to our calculation, the size of the array should be 20 bytes as int data type occupies 4 bytes, and array contains 5 elements,

so total memory space occupied by this array is $5 \times 4 = 20$ bytes. The same result has been shown by the **sizeof()** operator as we can observe in the following output.

Output



```
Size of the array 'arr' is : 20

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

Let's consider another scenario of an array.

```
#include <iostream>
using namespace std;
void fun(int arr[])
{
    std::cout << "Size of array is : " << sizeof(arr) << std::endl;
}
int main()
{
    int arr[]={10,20,30,40,50};
    fun(arr);
    return 0;
}
```

In the above program, we have tried to print the size of the array using the function. In this case, we have created an array of type integer, and we pass the 'arr' to the function **fun()**. The **fun()** would return the size of the integer pointer, i.e., **int***, and the size of the **int*** is 8 bytes in the 64-bit operating system.

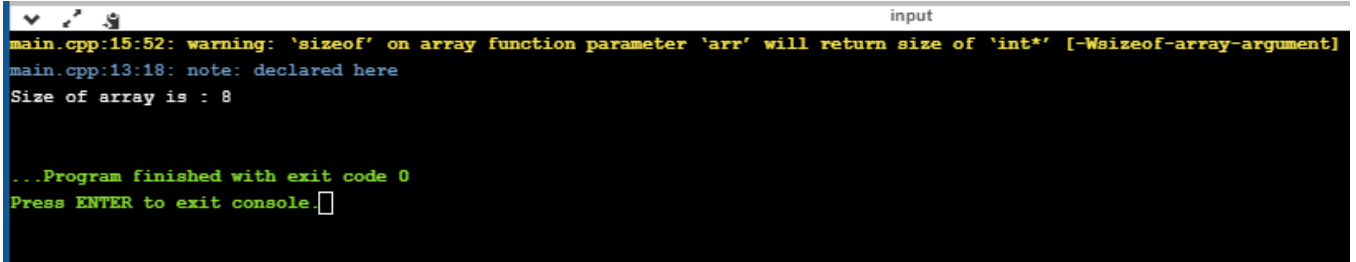
Output

AD

 **GAIAMTV**
TRANSFORMATION NETWORK

VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW



```

main.cpp:15:52: warning: 'sizeof' on array function parameter 'arr' will return size of 'int*' [-Wsizeof-array-argument]
main.cpp:13:18: note: declared here
Size of array is : 8

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

```

- When an operand is of pointer type.

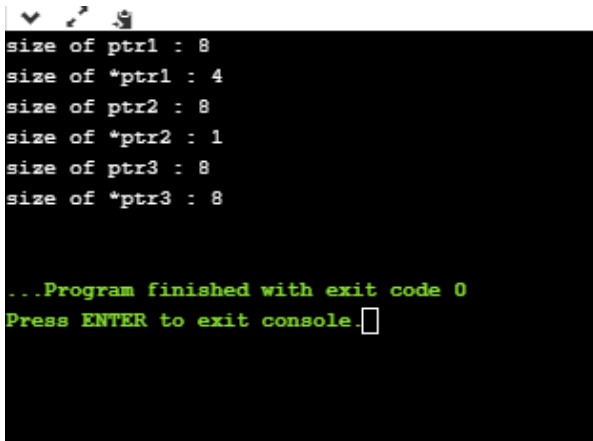
```

#include <iostream>
using namespace std;
int main()
{
    int *ptr1=new int(10);
    std::cout << "size of ptr1 : " <<sizeof(ptr1)
<< std::endl;
    std::cout << "size of *ptr1 : " <<sizeof(*ptr1)<< std::endl;
    char *ptr2=new char('a');
    std::cout <<"size of ptr2 : " <<sizeof(ptr2)<< std::endl;
    std::cout <<"size of *ptr2 : " <<sizeof(*ptr2)<< std::endl;
    double *ptr3=new double(12.78);
    std::cout <<"size of ptr3 : " <<sizeof(ptr3)<< std::endl;
    std::cout <<"size of *ptr3 : " <<sizeof(*ptr3)<< std::endl;
    return 0;
}

```

In the above program, we have determined the size of pointers. The size of pointers would remain same for all the data types. If the computer has 32bit operating system, then the size of the pointer would be 4 bytes. If the computer has 64-bit operating system, then the size of the pointer would be 8 bytes. I am running this program on 64-bit, so the output would be 8 bytes. Now, if we provide the '*' symbol to the pointer, then the output depends on the data type, for example, *ptr1 is of integer type means the sizeof() operator will return 4 bytes as int data type occupies 4 bytes.

Output



```
size of ptr1 : 8
size of *ptr1 : 4
size of ptr2 : 8
size of *ptr2 : 1
size of ptr3 : 8
size of *ptr3 : 8

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

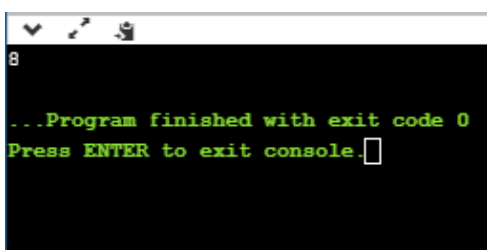
- **When an operand is an expression.**

```
#include <iostream>
using namespace std;

int main()
{
    int num1;
    double num2;
    cout << sizeof(num1+num2);
    return 0;
}
```

In the above program, we have declared two variables num1 and num2 of type int and double, respectively. The size of the int is 4 bytes, while the size of double is 8 bytes. The result would be the variable, which is of double type occupying 8 bytes.

Output



```
8

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

C++ Array of Pointers

Array and pointers are closely related to each other. In C++, the name of an array is considered as a pointer, i.e., the name of an array contains the address of an element. C++ considers the array name as the address of the first element. For example, if we create an array, i.e., marks which hold the 20 values of integer type, then marks will contain the address of first element, i.e., marks[0]. Therefore, we can say that array name (marks) is a pointer which is holding the address of the first element of an array.

Let's understand this scenario through an example.

```
#include <iostream>
using namespace std;
int main()
{
    int *ptr; // integer pointer declaration
    int marks[10]; // marks array declaration
    std::cout << "Enter the elements of an array : " << std::endl;
    for(int i=0;i<10;i++)
    {
        cin>>marks[i];
    }
    ptr=marks; // both marks and ptr pointing to the same element..
    std::cout << "The value of *ptr is : " <<*ptr<< std::endl;
    std::cout << "The value of *marks is : " <<*marks<<std::endl;
}
```

In the above code, we declare an integer pointer and an array of integer type. We assign the address of marks to the ptr by using the statement ptr=marks; it means that both the variables 'marks' and 'ptr' point to the same element, i.e., marks[0]. When we try to print the values of *ptr and *marks, then it comes out to be same. Hence, it is proved that the array name stores the address of the first element of an array.

Output



```
Enter the elements of an array :  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
The value of *ptr is :1  
The value of *marks is :1
```

Array of Pointers

An array of pointers is an array that consists of variables of pointer type, which means that the variable is a pointer addressing to some other element. Suppose we create an array of pointer holding 5 integer pointers; then its declaration would look like:

```
int *ptr[5];    // array of 5 integer pointer.
```

In the above declaration, we declare an array of pointer named as ptr, and it allocates 5 integer pointers in memory.

The element of an array of a pointer can also be initialized by assigning the address of some other element. Let's observe this case through an example.

```
int a; // variable declaration.  
ptr[2] = &a;
```

In the above code, we are assigning the address of 'a' variable to the third element of an array 'ptr'.

We can also retrieve the value of 'a' by dereferencing the pointer.

```
*ptr[2];
```

Let's understand through an example.

```
#include <iostream>
```



```
using namespace std;

int main()
{
    int ptr1[5]; // integer array declaration
    int *ptr2[5]; // integer array of pointer declaration

    std::cout << "Enter five numbers :> << std::endl;

    for(int i=0;i<5;i++)
    {
        std::cin >> ptr1[i];
    }

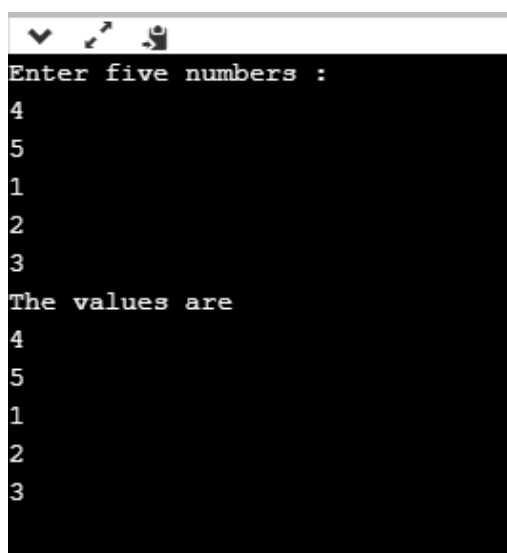
    for(int i=0;i<5;i++)
    {
        ptr2[i]=&ptr1[i];
    }

    // printing the values of ptr1 array
    std::cout << "The values are" << std::endl;

    for(int i=0;i<5;i++)
    {
        std::cout << *ptr2[i] << std::endl;
    }
}
```

In the above code, we declare an array of integer type and an array of integer pointers. We have defined the 'for' loop, which iterates through the elements of an array 'ptr1', and on each iteration, the address of element of ptr1 at index 'i' gets stored in the ptr2 at index 'i'.

Output



```
Enter five numbers :>
4
5
1
2
3
The values are
4
5
1
2
3
```

Till now, we have learnt the array of pointers to an integer. Now, we will see how to create the array of pointers to strings.

Array of Pointer to Strings

An array of pointer to strings is an array of character pointers that holds the address of the first character of a string or we can say the base address of a string.

The following are the differences between an array of pointers to string and two-dimensional array of characters:

- An array of pointers to string is more efficient than the two-dimensional array of characters in case of memory consumption because an array of pointer to strings consumes less memory than the two-dimensional array of characters to store the strings.
- In an array of pointers, the manipulation of strings is comparatively easier than in the case of 2d array. We can also easily change the position of the strings by using the pointers.

Let's see how to declare the array of pointers to string.

First, we declare the array of pointer to string:



```
char *names[5] = {"john",  
                  "Peter",  
                  "Marco",  
                  "Devin",  
                  "Ronan"};
```

In the above code, we declared an array of pointer names as 'names' of size 5. In the above case, we have done the initialization at the time of declaration, so we do not need to mention the size of the array of a pointer. The above code can be re-written as:

```
char *names[ ] = {"john",  
                  "Peter",  
                  "Marco",  
                  "Devin",  
                  "Ronan"};
```

In the above case, each element of the 'names' array is a string literal, and each string literal would hold the base address of the first character of a string. For example, names[0] contains the base address of "john", names[1] contains the base address of "Peter", and so on. It is not guaranteed that all the string literals will be stored in the contiguous memory location, but the characters of a string literal are stored in a contiguous memory location.

Let's create a simple example.

```
#include <iostream>
using namespace std;
int main()
{
    char *names[5] = {"john",
                     "Peter",
                     "Marco",
                     "Devin",
                     "Ronan"};
    for(int i=0;i<5;i++)
    {
        std::cout << names[i] << std::endl;
    }
    return 0;
}
```

In the above code, we have declared an array of char pointer holding 5 string literals, and the first character of each string is holding the base address of the string.

Output



A screenshot of a terminal window with a black background. It shows the output of the C++ program: 'john', 'Peter', 'Marco', 'Devin', and 'Ronan', each on a new line. The text is in a light blue or white monospaced font.

C++ Void Pointer

A void pointer is a general-purpose pointer that can hold the address of any data type, but it is not associated with any data type.

Syntax of void pointer

```
void *ptr;
```

In C++, we cannot assign the address of a variable to the variable of a different data type. Consider the following example:

```
int *ptr; // integer pointer declaration
float a=10.2; // floating variable initialization
ptr= &a; // This statement throws an error.
```

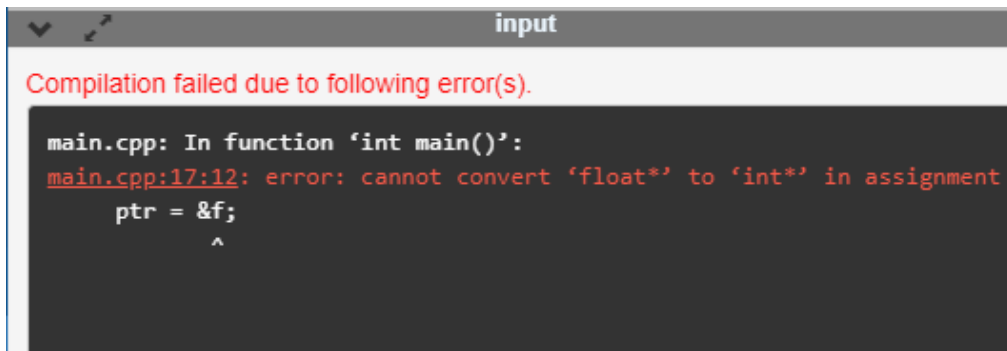
In the above example, we declare a pointer of type integer, i.e., ptr and a float variable, i.e., 'a'. After declaration, we try to store the address of 'a' variable in 'ptr', but this is not possible in C++ as the variable cannot hold the address of different data types.

Let's understand through a simple example.

```
#include <iostream.h>
using namespace std;
int main()
{
    int *ptr;
    float f=10.3;
    ptr = &f; // error
    std::cout << "The value of *ptr is : " <<*ptr<< std::endl;
    return 0;
}
```

In the above program, we declare a pointer of integer type and variable of float type. An integer pointer variable cannot point to the float variable, but it can point to an only integer variable.

Output



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

main.cpp: In function 'int main()':
main.cpp:17:12: error: cannot convert 'float*' to 'int*' in assignment
    ptr = &f;
           ^
```

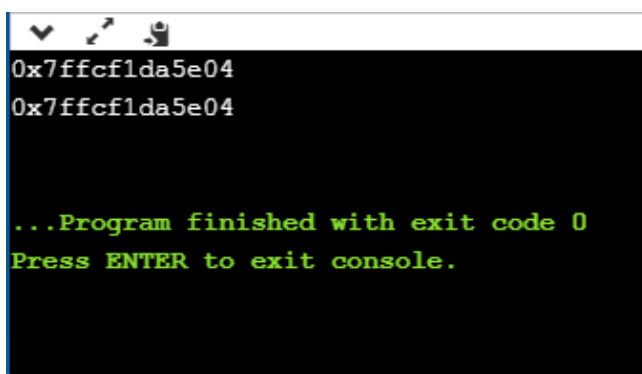
C++ has overcome the above problem by using the C++ void pointer as a void pointer can hold the address of any data type.

Let's look at a simple example of void pointer.

```
#include <iostream>
using namespace std;
int main()
{
    void *ptr; // void pointer declaration
    int a=9; // integer variable initialization
    ptr=&a; // storing the address of 'a' variable in a void pointer variable.
    std::cout << &a << std::endl;
    std::cout << ptr << std::endl;
    return 0;
}
```

In the above program, we declare a void pointer variable and an integer variable where the void pointer contains the address of an integer variable.

Output



```
0x7ffcfl1da5e04
0x7ffcfl1da5e04

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

Difference between void pointer in C and C++

In C, we can assign the void pointer to any other pointer type without any typecasting, whereas in C++, we need to typecast when we assign the void pointer type to any other pointer type.

Let's understand through a simple example.

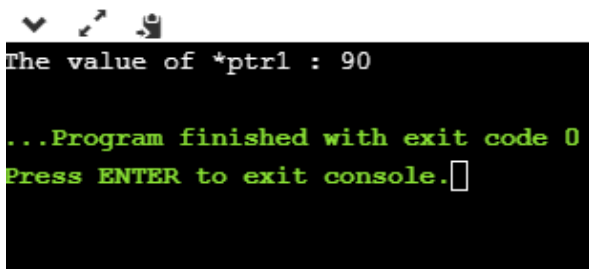
In C,

```
#include <stdio.h>

int main()
{
    void *ptr; // void pointer declaration
    int *ptr1; // integer pointer declaration
    int a = 90; // integer variable initialization
    ptr = &a; // storing the address of 'a' in ptr
    ptr1 = ptr; // assigning void pointer to integer pointer type.
    printf("The value of *ptr1 : %d", *ptr1);
    return 0;
}
```

In the above program, we declare two pointers 'ptr' and 'ptr1' of type void and integer, respectively. We also declare the integer type variable, i.e., 'a'. After declaration, we assign the address of 'a' variable to the pointer 'ptr'. Then, we assign the void pointer to the integer pointer, i.e., ptr1 without any typecasting because in C, we do not need to typecast while assigning the void pointer to any other type of pointer.

Output



```
The value of *ptr1 : 90

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

In C++,

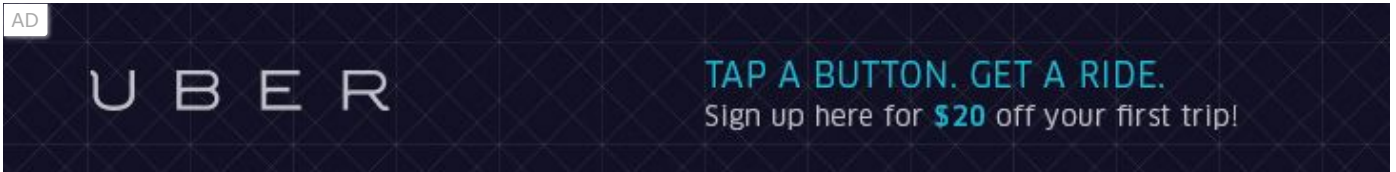
```
#include <iostream>

using namespace std;

int main()
```

```
{  
    void *ptr; // void pointer declaration  
    int *ptr1; // integer pointer declaration  
    int data=10; // integer variable initialization  
    ptr=&data; // storing the address of data variable in void pointer variable  
    ptr1=(int *)ptr; // assigning void pointer to integer pointer  
    std::cout << "The value of *ptr1 is : " <<*ptr1<< std::endl;  
    return 0;  
}
```

In the above program, we declare two pointer variables of type void and int type respectively. We also create another integer type variable, i.e., 'data'. After declaration, we store the address of variable 'data' in a void pointer variable, i.e., ptr. Now, we want to assign the void pointer to integer pointer, in order to do this, we need to apply the cast operator, i.e., (int *) to the void pointer variable. This cast operator tells the compiler which type of value void pointer is holding. For casting, we have to type the data type and * in a bracket like (char *) or (int *).



Output

```
▼ ↗ ⓘ  
The value of *ptr1 is : 10  
  
...Program finished with exit code 0  
Press ENTER to exit console.□
```

[< Prev](#)[Next >](#)

C++ References

Till now, we have read that C++ supports two types of variables:

- An ordinary variable is a variable that contains the value of some type. For example, we create a variable of type `int`, which means that the variable can hold the value of type integer.
- A pointer is a variable that stores the address of another variable. It can be dereferenced to retrieve the value to which this pointer points to.
- There is another variable that C++ supports, i.e., references. It is a variable that behaves as an alias for another variable.

How to create a reference?

Reference can be created by simply using an ampersand (&) operator. When we create a variable, then it occupies some memory location. We can create a reference of the variable; therefore, we can access the original variable by using either name of the variable or reference. For example,

```
int a=10;
```

Now, we create the reference variable of the above variable.

```
int &ref=a;
```

The above statement means that 'ref' is a reference variable of 'a', i.e., we can use the 'ref' variable in place of 'a' variable.

C++ provides two types of references:

- References to non-const values
- References as aliases



References to non-const values

It can be declared by using & operator with the reference type variable.


```
#include <iostream>

using namespace std;

int main()
{
    int a=10;
    int &value=a;
    std::cout << value << std::endl;
    return 0;
}
```

Output

10

References as aliases

References as aliases is another name of the variable which is being referenced.

For example,

```
int a=10; // 'a' is a variable.
int &b=a; // 'b' reference to a.
int &c=a; // 'c' reference to a.
```

Let's look at a simple example.

```
#include <iostream>

using namespace std;

int main()
{
    int a=70; // variable initialization
    int &b=a;
    int &c=a;
    std::cout << "Value of a is :" <<a<< std::endl;
    std::cout << "Value of b is :" <<b<< std::endl;
    std::cout << "Value of c is :" <<c<< std::endl;
    return 0;}
```

In the above code, we create a variable 'a' which contains a value '70'. We have declared two reference variables, i.e., b and c, and both are referring to the same variable 'a'. Therefore, we can say that 'a' variable can be accessed by 'b' and 'c' variable.

Output

```
Value of a is :70
Value of b is :70
Value of c is :70
```

Properties of References

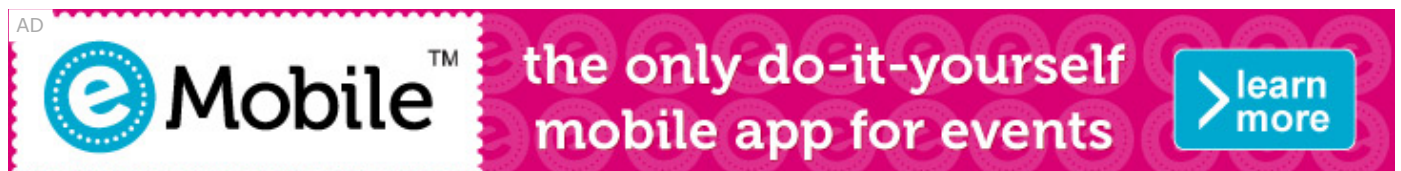
The following are the properties of references:

Initializátion

It must be initialized at the time of the declaration.

```
#include <iostream>
using namespace std;
int main()
{
    int a=10; // variable initialization
    int &b=a; // b reference to a
    std::cout << "value of a is " <<b<< std::endl;
    return 0;
}
```

In the above code, we have created a reference variable, i.e., 'b'. At the time of declaration, 'a' variable is assigned to 'b'. If we do not assign at the time of declaration, then the code would look like:



```
int &b;
&b=a;
```

The above code will throw a compile-time error as 'a' is not assigned at the time of declaration.

Output

```
value of a is 10
```

Reassignment

It cannot be reassigned means that the reference variable cannot be modified.

```
#include <iostream>
using namespace std;
int main()
{
    int x=11; // variable initialization
    int z=67;
    int &y=x; // y reference to x
    int &y=z; // y reference to z, but throws a compile-time error.
    return 0;}
```

In the above code, 'y' reference variable is referring to 'x' variable, and then 'z' is assigned to 'y'. But this reassignment is not possible with the reference variable, so it throws a compile-time error.



Compile-time error

```
main.cpp: In function 'int main()':
main.cpp:18:9: error: redeclaration of 'int& y'
    int &y=z; // y reference to z, but throws a compile-time error.
    ^
main.cpp:17:9: note: 'int& y' previously declared here
    int &y=x; // y reference to x
    ^
```

Function Parameters

References can also be passed as a function parameter. It does not create a copy of the argument and behaves as an alias for a parameter. It enhances the performance as it does not create a copy of the argument.

Let's understand through a simple example.

```
#include <iostream>
using namespace std;
int main()
{
    int a=9; // variable initialization
    int b=10; // variable initialization
    swap(a, b); // function calling
    std::cout << "value of a is :" <<a<< std::endl;
    std::cout << "value of b is :" <<b<< std::endl;
    return 0;
}
void swap(int &p, int &q) // function definition
{
    int temp; // variable declaration
    temp=p;
    p=q;
    q=temp;
}
```

In the above code, we are swapping the values of 'a' and 'b'. We have passed the variables 'a' and 'b' to the swap() function. In swap() function, 'p' is referring to 'a' and 'q' is referring to 'b'. When we swap the values of 'p' and 'q' means that the values of 'a' and 'b' are also swapped.

Output

```
value of a is :10
value of b is :9
```

References as shortcuts

With the help of references, we can easily access the nested data.

```
#include <iostream>
```

```
using namespace std;
struct profile
{
    int id;
};
struct employee
{
    profile p;
};
int main()
{
    employee e;
    int &ref=e.p.id;
    ref=34;
    std::cout << e.p.id << std::endl;
}
```

In the above code, we are trying to access the 'id' of the profile struct of the employee. We generally access this member by using the statement `e.p.id`, but this would be a tedious task if we have multiple access to this member. To avoid this situation, we create a reference variable, i.e., `ref`, which is another name of 'e.p.id'.

Output

34

[< Prev](#)[Next >](#)

C++ Reference vs Pointer

C++ reference and pointer seem to be similar, but there are some differences that exist between them. A reference is a variable which is another name of the existing variable, while the pointer is variable that stores the address of another variable.

What is Reference?

A reference is a **variable** that is referred to as another name for an already existing variable. The reference of a variable is created by storing the address of another variable.

A reference variable can be considered as a constant pointer with automatic indirection. Here, automatic indirection means that the compiler automatically applies the indirection operator (*).

Example of reference:

```
int &a = i;
```

In the above declaration, 'a' is an alias name for 'i' variable. We can also refer to the 'i' variable through 'a' variable also.

Let's understand through an example.

```
#include <iostream>
using namespace std;
int main()
{
    int i=8; // variable initialization
    int &a=i; // creating a reference variable
    cout<<"The value of 'i' variable is :"<<a;
    return 0;
}
```

In the above code, we have created a reference variable, i.e., 'a' for 'i' variable. After creating a reference variable, we can access the value of 'i' with the help of 'a' variable.

What is Pointer?

A pointer is a variable that contains the address of another variable. It can be dereferenced with the help of (*) **operator** to access the memory location to which the pointer points.

Differences between Reference and Pointer

The following are the differences between reference and pointer:

- **Definition**

A reference variable is another name for an already existing variable. It is mainly used in '**pass by reference**' where the reference variable is passed as a parameter to the function and the function to which this variable is passed works on the original copy of the variable.

Let's understand through a simple example.

```
#include <iostream>
using namespace std;
void func(int &);
int main()
{
    int a=10;
```

```
std::cout << "Value of 'a' is : " << a << std::endl;
func(a);
std::cout << "Now value of 'a' is : " << a << std::endl;
return 0;
}
void func(int &m)
{
    m=8;
}
```

Output:

```
Value of 'a' is :10
Now value of 'a' is :8
```

Whereas, **Pointer** is a variable that stores the address of another variable. It makes the programming easier as it holds the memory address of some variable.

- **Declaration**



We can declare a reference variable by adding a '&' symbol before a variable. If this symbol is used in the expression, then it will be treated as an address operator.

Before using a pointer variable, we should declare a pointer variable, and this variable is created by adding a '*' operator before a variable.

- **Reassignment**

We cannot reassign the reference variable. Now, we take a simple example as given below:

```
#include <iostream>
using namespace std;
void func(int &);
int main()
{
    int i; // variable declaration
    int k; // variable declaration
    int &a=i;
    int &a=k; // error
    return 0;
}
```

The above code shows the error that multiple declarations of **int &a** are not allowed. Therefore, the above program concludes that reassignment operation is not valid for the reference variable.



Whereas, the pointers can be re-assigned. This reassignment is useful when we are working with the data structures such as linked list, trees, etc.

- **Memory Address**

In the case of reference, both the reference and actual variable refer to the same address. The new variable will not be assigned to the reference variable until the actual variable is either deleted or goes out of the scope.

Let's understand this scenario through an example.

```
#include <iostream>
using namespace std;
void func(int &);
int main()
{
    int i;
    int &a=i;
    std::cout << "The address of 'a' variable is : " <<&a<< std::endl;
    std::cout << "The address of 'i' variable is : " <<&i<< std::endl;
    return 0;
}
```

Output:

```
The address of 'a' variable is : 0x7fff078e7e44
```

The above output shows that both the reference variable and the actual variable have the same address.



In the case of pointers, both the pointer variable and the actual variable will have different memory addresses. Let's understand this through an example.

```
#include <iostream>
using namespace std;
int main()
{
    int k;
    int *p;
    p=&k;
    cout<<"The memory address of p variable is : "<<&p;
    cout<<"\nThe memory address of k variable is : "<<&k;
    return 0;
}
```

Output:

```
The memory address of p variable is :0x7ffc5c164b8
```

The memor

- o **NULL value**

We cannot assign the NULL value to the reference variable, but the pointer variable can be assigned with a NULL value.

- o **Indirection**

Pointers can have pointer to pointer offering more than one level of indirection.

```
#include <iostream>
```



```
using namespace std;
int main()
{
    int *p;
    int a=8;
    int **q;
    p=&a;
    q=&p;
    std::cout << "The value of q is : " <<*q<< std::endl;
    return 0;
}
```

In the above code, the pointer 'p' is pointing to variable 'a' while 'q' is a double pointer which is pointing to 'p'. Therefore, we can say that the value of 'p' would be the address of 'a' variable and the value of 'q' variable would be the address of 'p' variable.

Output:

```
The value of q is : 0x7ffd104891dc
```

In the case of References, reference to reference is not possible. If we try to do **c++** program will throw a compile-time error

Let's understand this scenario through an example.

```
#include <iostream>
using namespace std;
int main()
{
    int a=8; // variable initialization
    int &p=a; // creating a reference variable for ?a? variable.
    int &&q=p; // reference to reference is not valid, it throws an error.
    return 0;
}
```

Output:

```
main.cpp: In function 'int main()':
main.cpp:18:10: error: cannot bind 'int' lvalue to 'int&&'
int &&q=p;
```

o Arithmetic Operations

As we know that arithmetic operations can be applied to the pointers named as "**Pointer Arithmetic**", but arithmetic operations cannot be applied on the references. There is no word, i.e., Reference Arithmetic exists in **C++**.

Let's see a simple example of Pointers.

```
#include <iostream>
using namespace std;
int main()
{
    int a[]={1,2,3,4,5}; // array initialization
    int *ptr; // pointer declaration
    ptr=a; assigning base address to pointer ptr.
    cout<<"The value of *ptr is : "<<*ptr;
```

```
ptr=ptr+1; // incrementing the value of ptr by 1.  
std::cout << "\nThe value of *ptr is: " <<*ptr<< std::endl;  
return 0;  
}
```

Output:

```
The value of *ptr is :1  
The value of *ptr is: 2
```

Let's understand the references through an example.

```
#include <iostream>  
using namespace std;  
int main()  
{  
  
    int value=90; // variable declaration  
    int &a=value; // assigning value to the reference  
    &a=&a+5 // arithmetic operation is not possible with reference variable, it throws an error.  
    return 0;  
}
```

The above code will throw a compile-time error as arithmetic operations are not allowed with references.

[< Prev](#)[Next >](#)

AD

 **For Videos Join Our Youtube Channel: [Join Now](#)**

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share

Learn Latest Tutorials

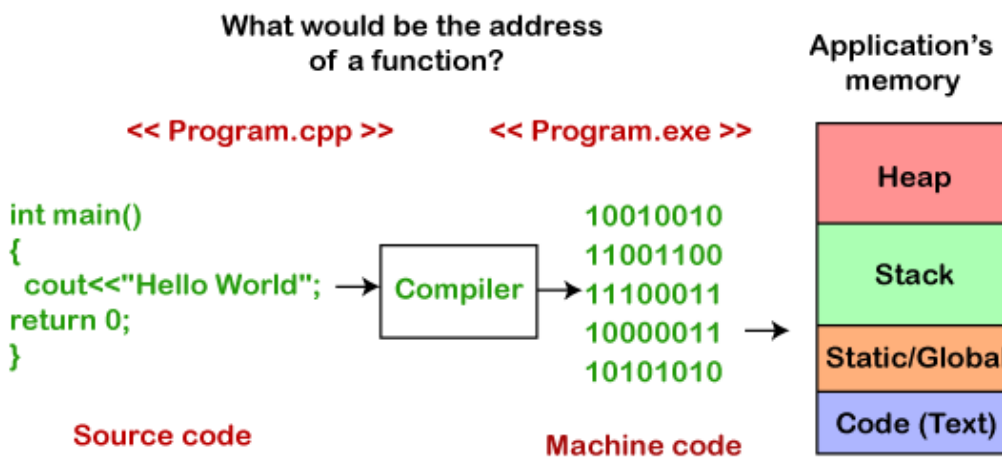
Function Pointer in C++

As we know that pointers are used to point some variables; similarly, the function pointer is a pointer used to point functions. It is basically used to store the address of a function. We can call the function by using the function pointer, or we can also pass the pointer to another function as a parameter.

They are mainly useful for event-driven applications, callbacks, and even for storing the functions in arrays.

What is the address of a function?

Function Pointers



Computer only understands the low-level language, i.e., binary form. The program we write in C++ is always in high-level language, so to convert the program into binary form, we use compiler. Compiler is a program that converts source code into an executable file. This executable file gets stored in RAM. The CPU starts the execution from the `main()` method, and it reads the copy in RAM but not the original file.

All the functions and machine code instructions are data. This data is a bunch of bytes, and all these bytes have some address in RAM. The function pointer contains RAM address of the first instruction of a function.

Syntax for Declaration

The following is the syntax for the declaration of a function pointer:

```
int (*FuncPtr) (int,int);
```

The above syntax is the function declaration. As functions are not simple as variables, but C++ is a type safe, so function pointers have return type and parameter list. In the above syntax, we first supply the return type, and then the name of the pointer, i.e., FuncPtr which is surrounded by the brackets and preceded by the pointer symbol, i.e., (*). After this, we have supplied the parameter list (int,int). The above function pointer can point to any function which takes two integer parameters and returns integer type value.

Address of a function

We can get the address of a function very easily. We just need to mention the name of the function, we do not need to call the function.

Let's illustrate through an example.

```
#include <iostream>
using namespace std;
int main()
{
    std::cout << "Address of a main() function is : " << &main << std::endl;
    return 0;
}
```

In the above program, we are displaying the address of a main() function. To print the address of a main() function, we have just mentioned the name of the function, there is no bracket not parameters. Therefore, the name of the function by itself without any brackets or parameters means the address of a function.

We can use the alternate way to print the address of a function, i.e., &main.

Calling a function indirectly

We can call the function with the help of a function pointer by simply using the name of the function pointer. The syntax of calling the function through the function pointer would be similar as we do the calling of the function normally.

Let's understand this scenario through an example.

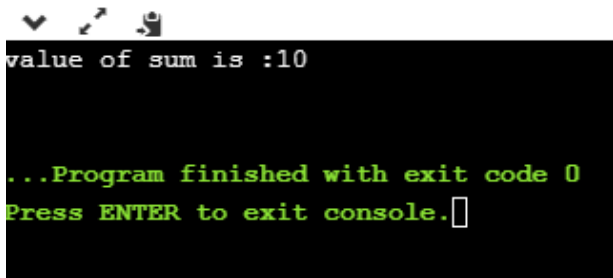
```
#include <iostream>
using namespace std;
int add(int a , int b)
```

```
{
    return a+b;
}

int main()
{
    int (*funcptr)(int,int); // function pointer declaration
    funcptr=add; // funcptr is pointing to the add function
    int sum=funcptr(5,5);
    std::cout << "value of sum is :" <<sum<< std::endl;
    return 0;
}
```

In the above program, we declare the function pointer, i.e., `int (*funcptr)(int,int)` and then we store the address of `add()` function in `funcptr`. This implies that `funcptr` contains the address of `add()` function. Now, we can call the `add()` function by using `funcptr`. The statement `funcptr(5,5)` calls the `add()` function, and the result of `add()` function gets stored in `sum` variable.

Output:



```
value of sum is :10

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

Let's look at another example of function pointer.

```
#include <iostream>
using namespace std;
void printname(char *name)
{
    std::cout << "Name is :" <<name<< std::endl;
}

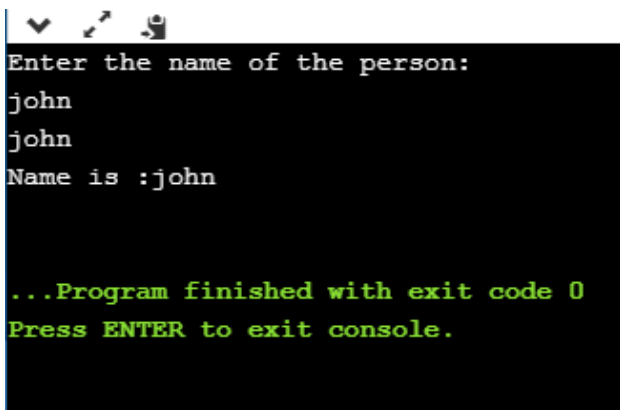
int main()
{
    char s[20]; // array declaration
    void (*ptr)(char*); // function pointer declaration
```

```
ptr=printname; // storing the address of printname in ptr.  
std::cout << "Enter the name of the person: " << std::endl;  
cin>>s;  
cout<<s;  
ptr(s); // calling printname() function  
return 0;  
}
```

In the above program, we define the function `printname()` which contains the char pointer as a parameter. We declare the function pointer, i.e., `void (*ptr)(char*)`. The statement `ptr=printname` means that we are assigning the address of `printname()` function to `ptr`. Now, we can call the `printname()` function by using the statement `ptr(s)`.

Output:

AD



```
Enter the name of the person:  
john  
john  
Name is :john  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

Passing a function pointer as a parameter

The function pointer can be passed as a parameter to another function.

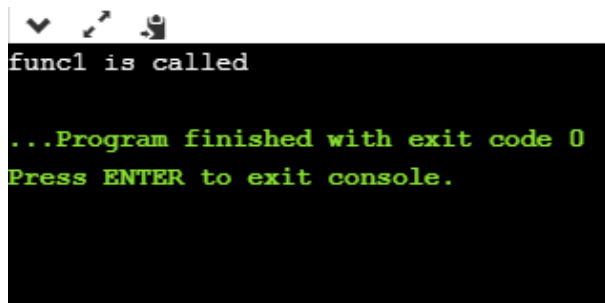
Let's understand through an example.

```
#include <iostream>  
using namespace std;  
void func1()  
{  
    cout<<"func1 is called";  
}  
void func2(void (*funcptr)())
```

```
{  
    funcptr();  
}  
  
int main()  
{  
    func2(func1);  
    return 0;  
}
```

In the above code, the func2() function takes the function pointer as a parameter. The main() method calls the func2() function in which the address of func1() is passed. In this way, the func2() function is calling the func1() indirectly.

Output:



```
func1 is called  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

[< Prev](#)[Next >](#)

AD

 Youtube **For Videos Join Our Youtube Channel: [Join Now](#)**

What is Memory Management?

Memory management is a process of managing computer memory, assigning the memory space to the programs to improve the overall system performance.

Why is memory management required?

As we know that arrays store the homogeneous data, so most of the time, memory is allocated to the array at the declaration time. Sometimes the situation arises when the exact memory is not determined until runtime. To avoid such a situation, we declare an array with a maximum size, but some memory will be unused. To avoid the wastage of memory, we use the new operator to allocate the memory dynamically at the run time.

Memory Management Operators

In C language, we use the **malloc()** or **calloc()** functions to allocate the memory dynamically at run time, and **free()** function is used to deallocate the dynamically allocated memory. C++ also supports these functions, but C++ also defines unary operators such as **new** and **delete** to perform the same tasks, i.e., allocating and freeing the memory.

New operator

A **new** operator is used to create the object while a **delete** operator is used to delete the object. When the object is created by using the new operator, then the object will exist until we explicitly use the delete operator to delete the object. Therefore, we can say that the lifetime of the object is not related to the block structure of the program.

Syntax

```
pointer_variable = new data-type
```

The above syntax is used to create the object using the new operator. In the above syntax, '**pointer_variable**' is the name of the pointer variable, '**new**' is the operator, and '**data-type**' defines the type of the data.

Example 1:

```
int *p;
```



```
p = new int;
```

In the above example, 'p' is a pointer of type int.

Example 2:

```
float *q;  
q = new float;
```

In the above example, 'q' is a pointer of type float.

In the above case, the declaration of pointers and their assignments are done separately. We can also combine these two statements as follows:

```
int *p = new int;  
float *q = new float;
```

Assigning a value to the newly created object

Two ways of assigning values to the newly created object:

- We can assign the value to the newly created object by simply using the assignment operator. In the above case, we have created two pointers 'p' and 'q' of type int and float, respectively. Now, we assign the values as follows:

```
*p = 45;  
*q = 9.8;
```

We assign 45 to the newly created int object and 9.8 to the newly created float object.

- We can also assign the values by using new operator which can be done as follows:



```
pointer_variable = new data-type(value);
```

Let's look at some examples.

```
int *p = new int(45);  
float *p = new float(9.8);
```

How to create a single dimensional array

As we know that new operator is used to create memory space for any data-type or even user-defined data type such as an array, structures, unions, etc., so the syntax for creating a one-dimensional array is given below:

```
pointer-variable = new data-type[size];
```

Examples:

```
int *a1 = new int[8];
```

In the above statement, we have created an array of type int having a size equal to 8 where p[0] refers first element, p[1] refers the first element, and so on.

Delete operator

When memory is no longer required, then it needs to be deallocated so that the memory can be used for another purpose. This can be achieved by using the delete operator, as shown below:

AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

```
delete pointer_variable;
```

In the above statement, '**delete**' is the operator used to delete the existing object, and '**pointer_variable**' is the name of the pointer variable.

In the previous case, we have created two pointers 'p' and 'q' by using the new operator, and can be deleted by using the following statements:

```
delete p;
```

```
delete q;
```

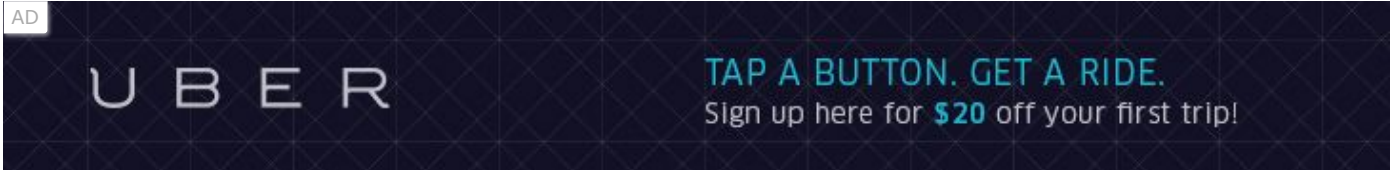
The dynamically allocated array can also be removed from the memory space by using the following syntax:

```
delete [size] pointer_variable;
```

In the above statement, we need to specify the size that defines the number of elements that are required to be freed. The drawback of this syntax is that we need to remember the size of the array. But, in recent versions of C++, we do not need to mention the size as follows:

```
delete [] pointer_variable;
```

Let's understand through a simple example:

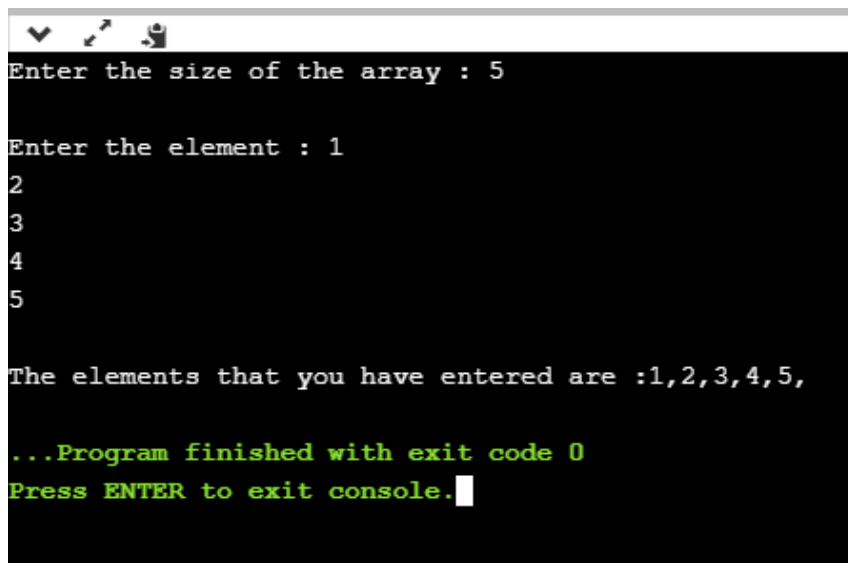


```
#include <iostream>
using namespace std
int main()
{
    int size; // variable declaration
    int *arr = new int[size]; // creating an array
    cout<<"Enter the size of the array : ";
    std::cin >> size; //
    cout<<"\nEnter the element : ";
    for(int i=0;i<size;i++) // for loop
    {
        cin>>arr[i];
    }
    cout<<"\nThe elements that you have entered are :";
    for(int i=0;i<size;i++) // for loop
    {
        cout<<arr[i]<<" ";
    }
```

```
}  
delete arr; // deleting an existing array.  
return 0;  
}
```

In the above code, we have created an array using the new operator. The above program will take the user input for the size of an array at the run time. When the program completes all the operations, then it deletes the object by using the statement **delete arr**.

Output



```
Enter the size of the array : 5  
  
Enter the element : 1  
2  
3  
4  
5  
  
The elements that you have entered are :1,2,3,4,5,  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

Advantages of the new operator

The following are the advantages of the new operator over malloc() function:

- It does not use the sizeof() operator as it automatically computes the size of the data object.
- It automatically returns the correct data type pointer, so it does not need to use the typecasting.
- Like other operators, the new and delete operator can also be overloaded.
- It also allows you to initialize the data object while creating the memory space for the object.

malloc() vs new in C++

Both the **malloc()** and **new** in C++ are used for the same purpose. They are used for allocating memory at the runtime. But, **malloc()** and **new** have different syntax. The main difference between the **malloc()** and **new** is that the **new** is an operator while **malloc()** is a standard library function that is predefined in a **stdlib** header file.

What is new?

The **new** is a memory allocation operator, which is used to allocate the memory at the runtime. The memory initialized by the **new** operator is allocated in a heap. It returns the starting address of the memory, which gets assigned to the variable. The functionality of the **new operator in C++** is similar to the **malloc()** function, which was used in the **C programming language**. **C++** is compatible with the **malloc()** function also, but the **new** operator is mostly used because of its advantages.

Syntax of new operator

```
type variable = new type(parameter_list);
```

In the above syntax

type: It defines the datatype of the variable for which the memory is allocated by the **new** operator.

variable: It is the name of the variable that points to the memory.

parameter_list: It is the list of values that are initialized to a variable.

The **new** operator does not use the **sizeof()** operator to allocate the memory. It also does not use the **resize** as the **new** operator allocates sufficient memory for an object. It is a construct that calls the constructor at the time of declaration to initialize an object.

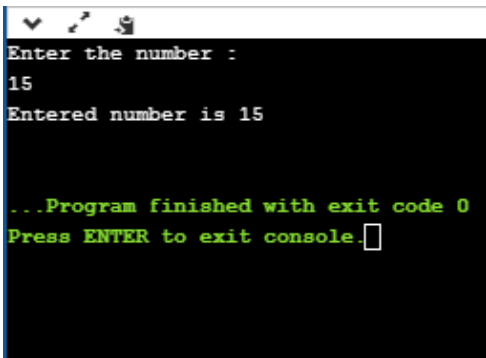
As we know that the **new** operator allocates the memory in a heap; if the memory is not available in a heap and the **new** operator tries to allocate the memory, then the exception is thrown. If our code is not able to handle the exception, then the program will be terminated abnormally.

Let's understand the new operator through an example.

```
#include <iostream>
using namespace std;
int main()
{
```

```
int *ptr; // integer pointer variable declaration
ptr=new int; // allocating memory to the pointer variable ptr.
std::cout << "Enter the number : " << std::endl;
std::cin >> *ptr;
std::cout << "Entered number is " << *ptr << std::endl;
return 0;
}
```

Output:



```
Enter the number :
15
Entered number is 15

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

What is malloc()?

A malloc() is a function that allocates memory at the runtime. This function returns the void pointer, which means that it can be assigned to any pointer type. This void pointer can be further typecast to get the pointer that points to the memory of a specified type.

The syntax of the malloc() function is given below:

```
type variable_name = (type *)malloc(sizeof(type));
```

where,

type: it is the datatype of the variable for which the memory has to be allocated.

variable_name: It defines the name of the variable that points to the memory.

(type*): It is used for typecasting so that we can get the pointer of a specified type that points to the memory.



sizeof(): The sizeof() operator is used in the malloc() function to obtain the memory size required for the allocation.

Note: The malloc() function returns the void pointer, so typecasting is required to assign a different type to the pointer. The sizeof() operator is required in the malloc() function as the malloc() function returns the raw memory, so the sizeof() operator will tell the malloc() function how much memory is required for the allocation.

If the sufficient memory is not available, then the memory can be resized using realloc() function. As we know that all the dynamic memory requirements are fulfilled using heap memory, so malloc() function also allocates the memory in a heap and returns the pointer to it. The heap memory is very limited, so when our code starts execution, it marks the memory in use, and when our code completes its task, then it frees the memory by using the free() function. If the sufficient memory is not available, and our code tries to access the memory, then the malloc() function returns the NULL pointer. The memory which is allocated by the malloc() function can be deallocated by using the free() function.

Let's understand through an example.

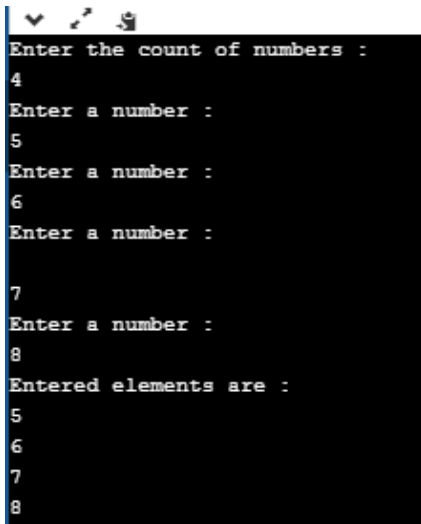
```
#include <iostream>
#include<stdlib.h>
using namespace std;

int main()
{

    int len; // variable declaration
    std::cout << "Enter the count of numbers : " << std::endl;
    std::cin >> len;
    int *ptr; // pointer variable declaration
    ptr=(int*) malloc(sizeof(int)*len); // allocating memory to the pointer variable
    for(int i=0;i<len;i++)
    {
        std::cout << "Enter a number : " << std::endl;
        std::cin >> *(ptr+i);
    }
    std::cout << "Entered elements are : " << std::endl;
    for(int i=0;i<len;i++)
```

```
{  
    std::cout << *(ptr+i) << std::endl;  
}  
free(ptr);  
    return 0;  
}
```


Output:




```
Enter the count of numbers :  
4  
Enter a number :  
5  
Enter a number :  
6  
Enter a number :  
7  
Enter a number :  
8  
Entered elements are :  
5  
6  
7  
8
```

If we do not use the **free()** function at the correct place, then it can lead to the cause of the dangling pointer. **Let's understand this scenario through an example.**

AD

 **eMobile™**

the only do-it-yourself
mobile app for events



```
#include <iostream>  
#include<stdlib.h>  
using namespace std;  
int *func()  
{  
    int *p;  
    p=(int*) malloc(sizeof(int));  
    free(p);  
    return p;  
}  
int main()  
{
```

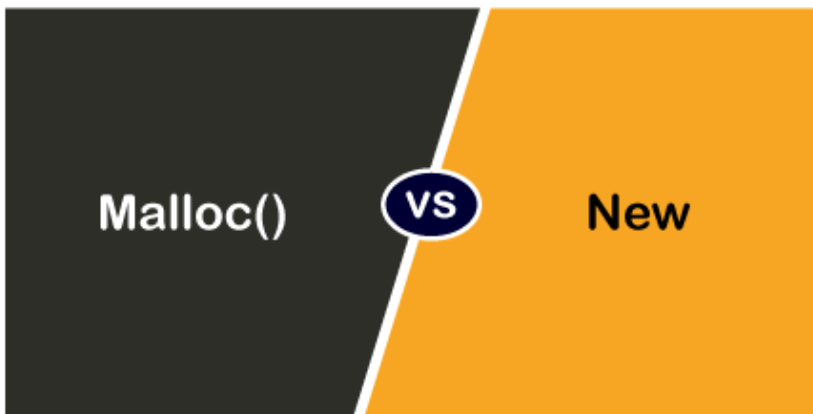


```
int *ptr;  
ptr=func();  
free(ptr);  
return 0;  
}
```

In the above code, we are calling the func() function. The func() function returns the integer pointer. Inside the func() function, we have declared a *p pointer, and the memory is allocated to this pointer variable using malloc() function. In this case, we are returning the pointer whose memory is already released. The ptr is a dangling pointer as it is pointing to the released memory location. Or we can say ptr is referring to that memory which is not pointed by the pointer.

Till now, we get to know about the new operator and the malloc() function. Now, we will see the differences between the new operator and the malloc() function.

Differences between the malloc() and new



- The new operator constructs an object, i.e., it calls the constructor to initialize an object while **malloc()** function does not call the constructor. The new operator invokes the constructor, and the delete operator invokes the destructor to destroy the object. This is the biggest difference between the malloc() and new.
- The new is an operator, while malloc() is a predefined function in the stdlib header file.
- The operator new can be overloaded while the malloc() function cannot be overloaded.
- If the sufficient memory is not available in a heap, then the new operator will throw an exception while the malloc() function returns a NULL pointer.
- In the new operator, we need to specify the number of objects to be allocated while in malloc() function, we need to specify the number of bytes to be allocated.
- In the case of a new operator, we have to use the delete operator to deallocate the memory. But in the case of malloc() function, we have to use the free() function to deallocate the

memory.

Syntax of new operator

```
type reference_variable = new type name;
```

where,

type: It defines the data type of the reference variable.

reference_variable: It is the name of the pointer variable.

new: It is an operator used for allocating the memory.

type name: It can be any basic data type.

For example,

```
int *p;  
p = new int;
```

In the above statements, we are declaring an integer pointer variable. The statement **p = new int;** allocates the memory space for an integer variable.

Syntax of malloc() is given below:

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

ptr: It is a pointer variable.

data_type: It can be any basic data type.

For example,

```
int *p;  
p = (int *) malloc(sizeof(int))
```

The above statement will allocate the memory for an integer variable in a heap, and then stores the address of the reserved memory in 'p' variable.

- On the other hand, the memory allocated using malloc() function can be deallocated using the free() function.
- Once the memory is allocated using the new operator, then it cannot be resized. On the other hand, the memory is allocated using malloc() function; then, it can be reallocated using realloc() function.
- The execution time of new is less than the malloc() function as new is a construct, and malloc is a function.
- The new operator does not return the separate pointer variable; it returns the address of the newly created object. On the other hand, the malloc() function returns the void pointer which can be further typecast in a specified type.

[< Prev](#)[Next >](#)

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

free vs delete in C++

In this topic, we are going to learn about the **free()** function and **delete** operator in C++.

free() function

The free() function is used in C++ to de-allocate the memory dynamically. It is basically a library function used in C++, and it is defined in **stdlib.h** header file. This library function is used when the pointers either pointing to the memory allocated using malloc() function or Null pointer.

Syntax of free() function

Suppose we have declared a pointer 'ptr', and now, we want to de-allocate its memory:

```
free(ptr);
```

The above syntax would de-allocate the memory of the pointer variable 'ptr'.

free() parameters

In the above syntax, ptr is a parameter inside the free() function. The ptr is a pointer pointing to the memory block allocated using malloc(), calloc() or realloc function. This pointer can also be null or a pointer allocated using malloc but not pointing to any other memory block.

- If the pointer is null, then the free() function will not do anything.
- If the pointer is allocated using malloc, calloc, or realloc, but not pointing to any memory block then this function will cause undefined behavior.

free() Return Value

The free() function does not return any value. Its main function is to free the memory.

Let's understand through an example.

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    int *ptr;
```

```
ptr = (int*) malloc(5*sizeof(int));
cout << "Enter 5 integer" << endl;

for (int i=0; i<5; i++)
{
    // *(ptr+i) can be replaced by ptr[i]
    cin >> ptr[i];
}
cout << endl << "User entered value" << endl;

for (int i=0; i<5; i++)
{
    cout << *(ptr+i) << " ";
}
free(ptr);

/* prints a garbage value after ptr is free */
cout << "Garbage Value" << endl;

for (int i=0; i<5; i++)
{
    cout << *(ptr+i) << " ";
}
return 0;
}
```

The above code shows how free() function works with malloc(). First, we declare integer pointer *ptr, and then we allocate the memory to this pointer variable by using malloc() function. Now, ptr is pointing to the uninitialized memory block of 5 integers. After allocating the memory, we use the free() function to destroy this allocated memory. When we try to print the value, which is pointed by the ptr, we get a garbage value, which means that memory is de-allocated.

Output



```

1
2
3
4
5

User entered value
1 2 3 4 5 Garbage Value
0 0 3 4 5

```

Let's see how free() function works with a calloc.

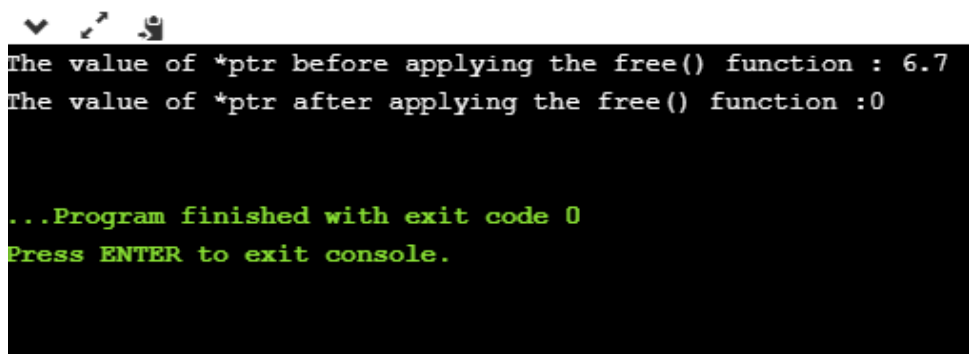
```

#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    float *ptr; // float pointer declaration
    ptr=(float*)calloc(1,sizeof(float));
    *ptr=6.7;
    std::cout << "The value of *ptr before applying the free() function : " <<*ptr<< std::endl;
    free(ptr);
    std::cout << "The value of *ptr after applying the free() function : " <<*ptr<< std::endl;
    return 0;
}

```

In the above example, we can observe that free() function works with a calloc(). We use the calloc() function to allocate the memory block to the float pointer ptr. We have assigned a memory block to the ptr that can have a single float type value.

Output:



```

The value of *ptr before applying the free() function : 6.7
The value of *ptr after applying the free() function : 0

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

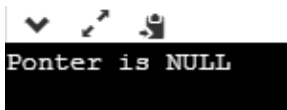
```

Let's look at another example.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    int *ptr1=NULL;
    int *ptr2;
    int x=9;
    ptr2=&x;
    if(ptr1)
    {
        std::cout << "Pointer is not Null" << std::endl;
    }
    else
    {
        cout<<"Ponter is NULL";
    }
    free(ptr1);
    //free(ptr2); // If this statement is executed, then it gives a runtime error.
    return 0;
}
```

The above code shows how free() function works with a NULL pointer. We have declared two pointers, i.e., ptr1 and ptr2. We assign a NULL value to the pointer ptr1 and the address of x variable to pointer ptr2. When we apply the free(ptr1) function to the ptr1, then the memory block assigned to the ptr is successfully freed. The statement free(ptr2) shows a runtime error as the memory block assigned to the ptr2 is not allocated using malloc or calloc function.

Output

A screenshot of a terminal window with a black background. At the top, there are three small icons: a downward arrow, a leftward arrow, and a magnifying glass. Below these icons, the text "Ponter is NULL" is displayed in a white, monospaced font.

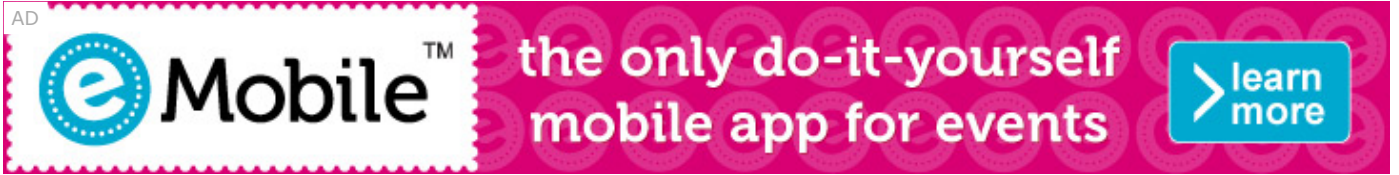
Delete operator

It is an operator used in C++ programming language, and it is used to de-allocate the memory dynamically. This operator is mainly used either for those pointers which are allocated using a new operator or NULL pointer.

Syntax

```
delete pointer_name
```

For example, if we allocate the memory to the pointer using the new operator, and now we want to delete it. To delete the pointer, we use the following statement:



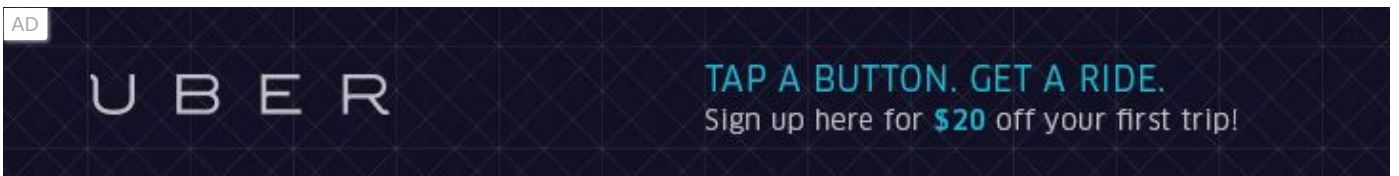
```
delete p;
```

To delete the array, we use the statement as given below:

```
delete [] p;
```

Some important points related to delete operator are:

- It is either used to delete the array or non-array objects which are allocated by using the new keyword.
- To delete the array or non-array object, we use delete[] and delete operator, respectively.
- The new keyword allocated the memory in a heap; therefore, we can say that the delete operator always de-allocates the memory from the heap
- It does not destroy the pointer, but the value or the memory block, which is pointed by the pointer is destroyed.



Let's look at the simple example of a delete operator.

```
#include <iostream>
#include <cstdlib>
using namespace std;

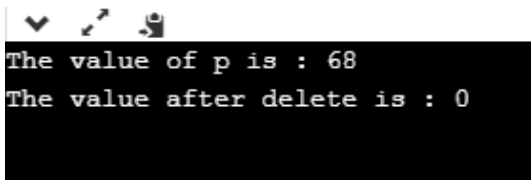
int main()
```



```
{  
    int *ptr;  
    ptr=new int;  
    *ptr=68;  
    std::cout << "The value of p is : " <<*ptr<< std::endl;  
    delete ptr;  
    std::cout <<"The value after delete is : " <<*ptr<< std::endl;  
    return 0;  
}
```

In the above code, we use the new operator to allocate the memory, so we use the delete ptr operator to destroy the memory block, which is pointed by the pointer ptr.

Output



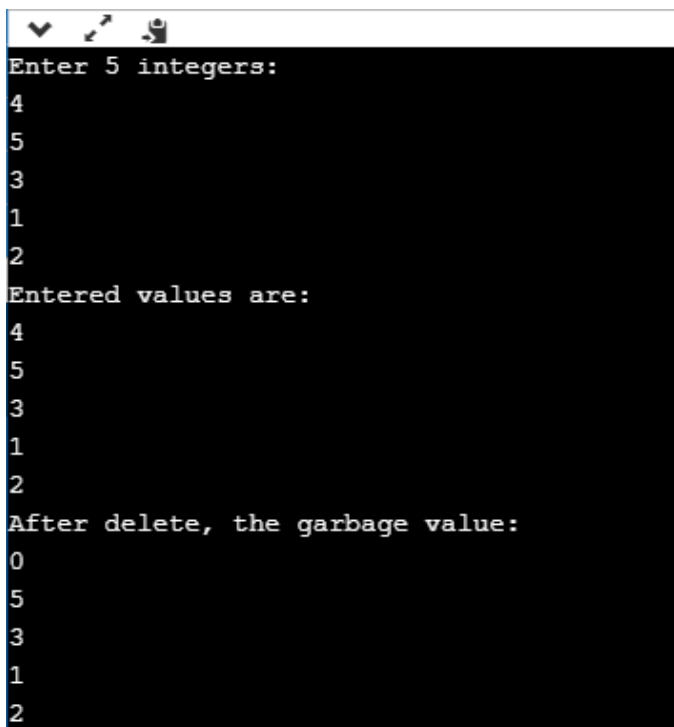
```
✓ ↗ 📄  
The value of p is : 68  
The value after delete is : 0
```

Let's see how delete works with an array of objects.

```
#include <iostream>  
using namespace std;  
int main()  
{  
    int *ptr=new int[5]; // memory allocation using new operator.  
    std::cout << "Enter 5 integers:" << std::endl;  
    for(int i=1;i<=5;i++)  
    {  
        cin>>ptr[i];  
    }  
    std::cout << "Entered values are:" << std::endl;  
    for(int i=1;i<=5;i++)  
    {  
        cout<<*(ptr+i)<<endl;  
    }  
}
```

```
}  
  
delete[] ptr; // deleting the memory block pointed by the ptr.  
std::cout << "After delete, the garbage value:" << std::endl;  
    for(int i=1;i<=5;i++)  
    {  
        cout<<*(ptr+i)<<endl;  
    }  
return 0;  
}
```

Output



The screenshot shows the output of a C++ program. It starts with the prompt "Enter 5 integers:" followed by five lines of input: 4, 5, 3, 1, and 2. Then, it displays "Entered values are:" followed by the same five numbers. Finally, it shows "After delete, the garbage value:" followed by five lines of output: 0, 5, 3, 1, and 2. The output demonstrates that after deleting the memory, the values are no longer the same as the input, indicating memory corruption or garbage values.

Differences between delete and free()

The following are the differences between delete and free() in C++ are:

- The delete is an operator that de-allocates the memory dynamically while the free() is a function that destroys the memory at the runtime.
- The delete operator is used to delete the pointer, which is either allocated using new operator or a NULL pointer, whereas the free() function is used to delete the pointer that is either allocated using malloc(), calloc() or realloc() function or NULL pointer.
- When the delete operator destroys the allocated memory, then it calls the destructor of the class in C++, whereas the free() function does not call the destructor; it only frees the memory from the heap.

- The delete() operator is faster than the free() function.

[< Prev](#)[Next >](#)

AD

 [For Videos Join Our Youtube Channel: Join Now](#)


Feedback


- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share





Learn Latest Tutorials


 [Splunk tutorial](#)
Splunk


 [SPSS tutorial](#)
SPSS


 [Swagger tutorial](#)
Swagger

 [T-SQL tutorial](#)
Transact-SQL

 [Tumblr tutorial](#)
Tumblr

 [React tutorial](#)
ReactJS

 [Regex tutorial](#)
Regex

 [Reinforcement learning tutorial](#)