**CPP-Day-3**

**Inline functions:**

Inside the main() method, when the function fun1() is called, the control is transferred to the definition of the called function. The addresses from where the function is called and the definition of the function are different. This control transfer takes a lot of time and increases the overhead.

When the inline function is encountered, then the definition of the function is copied to it. In this case, there is no control transfer which saves a lot of time and also decreases the overhead.

If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

To inline a function, place the keyword inline before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the inline specifier.

#include <iostream>

using namespace std;

inline int Max(int x, int y) {

   return (x > y)? x : y;

}

// Main function for the program

int main() {

   cout << "Max (20,10): " << Max(20,10) << endl;

   cout << "Max (0,200): " << Max(0,200) << endl;

   cout << "Max (100,1010): " << Max(100,1010) << endl;

   return 0;

}

Ex:

#include <iostream>

using namespace std;

inline int add(int a, int b)

{

    return(a+b);

}

int main()

{

    cout<<"Addition of 'a' and 'b' is:"<<add(2,3);A

    return 0;

}

Uses:

The main use of the inline function in C++ is to save memory space. Whenever the function is called, then it takes a lot of time to execute the tasks, such as moving to the calling function. If the length of the function is small, then the substantial amount of execution time is spent in such overheads, and sometimes time taken required for moving to the calling function will be greater than the time taken required to execute that function.

The solution to this problem is to use macro definitions known as macros. The preprocessor macros are widely used in C, but the major drawback with the macros is that these are not normal functions which means the error checking process will not be done during the compilation.

C++ has provided one solution to this problem. In the case of function calling, the time for calling such small functions is huge, so to overcome such a problem, a new concept was introduced known as an inline function. When the function is encountered inside the main() method, it is expanded with its definition thus saving time.

We cannot provide the inlining to the functions in the following circumstances:

If a function is recursive.

If a function contains a loop like for, while, do-while loop.

If a function contains static variables.

If a function contains a switch or go to statement

An inline function can be used in the following scenarios:

An inline function can be used when the performance is required.

It can be used over the macros.

We can use the inline function outside the class so that we can hide the internal implementation of the function.

Advantages of inline function

In the inline function, we do not need to call a function, so it does not cause any overhead.

It also saves the overhead of the return statement from a function.

It does not require any stack on which we can push or pop the variables as it does not perform any function calling.

An inline function is mainly beneficial for the embedded systems as it yields less code than a normal function.

Disadvantages of inline function

The following are the disadvantages of an inline function:

The variables that are created inside the inline function will consume additional registers. If the variables increase, then the use of registers also increases, which may increase the overhead on register variable resource utilization. It means that when the function call is replaced with an inline function body, then the number of variables also increases, leading to an increase in the number of registers. This causes an overhead on resource utilization.

If we use many inline functions, then the binary executable file also becomes large.

The use of so many inline functions can reduce the instruction cache hit rate, reducing the speed of instruction fetch from the cache memory to that of the primary memory.

It also increases the compile-time overhead because whenever the changes are made inside the inline function, then the code needs to be recompiled again to reflect the changes; otherwise, it will execute the old functionality.

Sometimes inline functions are not useful for many embedded systems because, in some cases, the size of the embedded is considered more important than the speed.

It can also cause thrashing due to the increase in the size of the binary executable file. If the thrashing occurs in the memory, then it leads to the degradation in the performance of the computer.

**Return from void functions in C++**

Void functions are “void” due to the fact that they are not supposed to return values. True, but not completely. We cannot return values but there is something we can surely return from void functions.

A void function can do return

We can simply write return statement in a void fun(). In-fact it is considered a good practice (for readability of code) to write return; statement to indicate end of function.

#include <iostream>

using namespace std;

void fun()

{

cout << "Hello";

// We can write return in void

return;

}

int main()

{

fun();

return 0;

}

Ex:

**A void fun() can return another void function**

// C++ code to demonstrate void()

// returning void()

#include<iostream>

using namespace std;

// A sample void function

void work()

{

    cout << "The void function has returned "

            " a void() !!! \n";

}

// Driver void() returning void work()

void test()

{

    // Returning void function

    return work();

}

int main()

{

    // Calling void function

    test();

    return 0;

}

A void() can return a void value.

A void() cannot return a value that can be used. But it can return a value which is void without giving an error.

Ex:

// C++ code to demonstrate void()

// returning a void value

#include<iostream>

using namespace std;

// Driver void() returning a void value

void test()

{

    cout << "Hello";

    // Returning a void value

    return (void)"Doesn't Print";

}

int main()

{

    test();

    return 0;

}

o/p: Hello

**Pointers in CPP**

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.

#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

C++ Void Pointer

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 \*).

**Exception handling**

Exception Handling in C++ is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors.

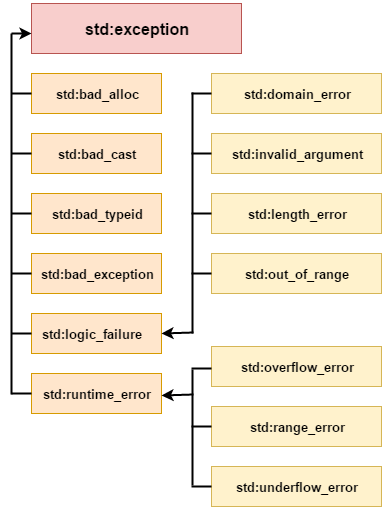
In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from std::exception class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

Advantage

It maintains the normal flow of the application. In such case, rest of the code is executed even after exception.

C++ Exception Classes

In C++ standard exceptions are defined in <exception> class that we can use inside our programs. The arrangement of parent-child class hierarchy is shown below:



All the exception classes in C++ are derived from std::exception class. Let's see the list of C++ common exception classes.

Exception Description

std::exception It is an exception and parent class of all standard C++ exceptions.

std::logic\_failure It is an exception that can be detected by reading a code.

std::runtime\_error It is an exception that cannot be detected by reading a code.

std::bad\_exception It is used to handle the unexpected exceptions in a c++ program.

std::bad\_cast This exception is generally be thrown by dynamic\_cast.

std::bad\_typeid This exception is generally be thrown by typeid.

std::bad\_alloc This exception is generally be thrown by new.

C++ Exception Handling Keywords

In C++, we use 3 keywords to perform exception handling:

try

catch, and

throw

C++ try/catch

In C++ programming, exception handling is performed using try/catch statement. The C++ try block is used to place the code that may occur exception. The catch block is used to handle the exception.

C++ example without try/catch

#include <iostream>

using namespace std;

float division(int x, int y) {

return (x/y);

}

int main () {

int i = 50;

int j = 0;

float k = 0;

k = division(i, j);

cout << k << endl;

return 0;

}

Output:

Floating point exception (core dumped)

C++ try/catch example

#include <iostream>

using namespace std;

float division(int x, int y) {

if( y == 0 ) {

throw "Attempted to divide by zero!";

}

return (x/y);

}

int main () {

int i = 25;

int j = 0;

float k = 0;

try {

k = division(i, j);

cout << k << endl;

}

catch (const char\* e)

{

cerr << e << endl;

}

return 0;

}

Output:

(or)

Attempted to divide by zero!

#include<iostream>

using namespace std;

float divide(int x,int y)

{

    if(y==0)

        throw "Div by zero is impossible";

    return x/y;

}

int main()

{

int a,b;

cout<<"Enter two numbers:";

fflush(stdin);

cin>>a>>b;

try

{

    float res=divide(a,b);

    cout<<"Result="<<res<<endl;

}

catch(const char\* e)

{

    cout<<e<<endl;

}

return 0;

}

Ex:

/\* Write a program to illustrate array index out of bounds exception. \*/

#include <iostream>

using namespace std;

int main() {

int a[5]={1,2,3,4,5},i;

try{

i=0;

while(1){

if(i!=5)

{

cout<<a[i]<<endl;

i++;

}

else

throw i;

}

}

catch(int i)

{

cout<<"Array Index out of Bounds Exception: "<<i<<endl;

}

return 0;

}

Ex:

/\* Write a C++ program to define function that generates exception. \*/

#include<iostream>

using namespace std;

void sqr()

{

int s;

cout<<"\n Enter a number:";

cin>>s;

if (s>0)

{

cout<<"Square="<<s\*s;

}

else

{

throw (s);

}

}

int main()

{

try

{

sqr();

}

catch (int j)

{

cout<<"\n Caught the exception \n";

}

return 0;

}

Multiple catch Statements

It is also possible that a program segment has more than one condition to throw an

exception. In such cases, we can associate more than one catch statement with a try (similar to

switch statement). The format of multiple catch statements is as follows:

try

{

// try block

}

catch (type1 arg)

{

// catch section1

}

catch (type2 arg)

{

// catch section2

}

. . . . . . .

. . . . . . .

catch (typen arg)

{

// catch section-n

}

When an exception is thrown, the exception handlers are searched in the order for an

appropriate mach. The first handler that yields a match is executed. After executing the handler,

the control goes to the first statement after the last catch block for that try. When no match is

found the program will terminate.

It is possible that arguments of several catch statements match the type of exception. In

such cases, the first handler that matches the exception type is executed.

/\*Write a C++ program to throw multiple exceptions and define multiple catch

statement. \*/

#include <iostream>

using namespace std;

void num (int k)

{

try

{

if (k==0) throw k;

else

if (k>0) throw 'P';

else

if (k<0) throw 1.0;

cout<<"\*\*\* end of try block \*\*\*\n";

}

catch(char g)

{

cout<<"Caught a positive value \n";

}

catch (int j)

{

cout<<"caught an null value \n";

}

catch (double f)

{

cout<<"Caught a Negative value \n";

}

cout<<"\*\*\* end of try catch \*\*\*\n \n";

}

int main()

{

cout<<"Demo of Multiple catches"<<endl;

num(0);

num(5);

num(-1);

return 0;

}

Catching Multiple Exceptions

In some situations, we may not able to anticipate all possible types of exceptions and therefore

may not be able to design independent catch handlers to catch them. In such circumstances, we

can create a catch statement to catch all exceptions instead of a certain type alone.

Syntax:

catch(...)

{

// Statements for handling

// all exceptions

}

/\* Write a C++ program to catch multiple exceptions.\*/

#include <iostream>

using namespace std;

void num (int k)

{

try

{

if (k==0) throw k;

else

if (k>0) throw 'P';

else

if (k<0) throw 1.0;

}

catch(...)

{

cout<<"Caught an Exception"<<endl;

}

}

int main()

{

cout<<"Demo of Multiple catches"<<endl;

num(0);

num(5);

num(-1);

return 0;

}

Specifying Exceptions

It is possible to restrict a function to throw only certain specified exceptions. This is

achieved by adding a throw list clause to the function definition. The general form of using an

exception specification is:

return\_type fucntion\_name (parameter list) throw (data type list)

{

// function body

}

The data type list indicates the type of exception that is permitted to be thrown. If we

want to deny a function from throwing any exception, declaring the data type list void as per the

following statement can do this.

throw(); // void or vacant list

/\* Write a C++ program to restrict a function to throw only specified type of

exceptions. \*/

#include<iostream>

using namespace std;

void check (int k) throw(int)

{

if (k==1) throw 'k';

else

if (k==2) throw k;

else

if (k==-2) throw 1.0;

}

int main()

{

try {

check(1);

check(-2);

check(3);

}

catch (char g)

{

cout<<"Caught a character exception \n";

}

catch (int j)

{

cout<<"Caught a character exception \n";

}

catch (double s)

{

cout<<"Caught a double exception \n";

}

cout<<"\n End of main()";

return 0;

}

**C++ User-Defined Exceptions**

The new exception can be defined by overriding and inheriting exception class functionality.

C++ user-defined exception example

Let's see the simple example of user-defined exception in which std::exception class is used to define the exception.

#include <iostream>

#include <exception>

using namespace std;

class MyException : public exception{

public:

const char \* what() const throw()

{

return "Attempted to divide by zero!\n";

}

};

int main()

{

try

{

int x, y;

cout << "Enter the two numbers : \n";

cin >> x >> y;

if (y == 0)

{

MyException z;

throw z;

}

else

{

cout << "x / y = " << x/y << endl;

}

}

catch(exception& e)

{

cout << e.what();

}

}

Output:

Enter the two numbers :

10

2

x / y = 5

Enter the two numbers :

10

0

Attempted to divide by zero!

Dynamic memory Allocation

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);

int \*p = new int(45);

float \*p = new float(9.8);

Ex:

#include<iostream>

using namespace std;

int main()

{

int \*ptr=new int;

\*ptr=45;

cout<<\*ptr;

}

**Creating a single dimensional array:**

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:

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;

#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]<<",";

}

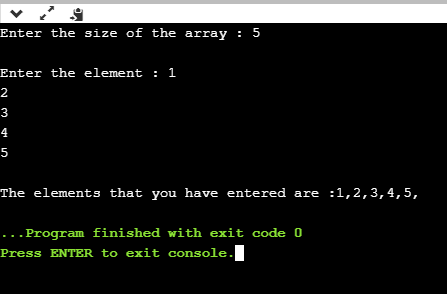
delete arr; // deleting an existing array.

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**



(or)

#include<iostream>

using namespace std;

int main()

{

int n;

int \*arr=new int[n];

cout<<"Enter n:";

cin>>n;

cout<<"Enter array elements:";

for(int i=0;i<n;i++)

    cin>>arr[i];

cout<<"Array is\n";

for(int i=0;i<n;i++)

    cout<<arr[i]<<" ";

delete(arr);

}

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.

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.

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.

ree() 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.

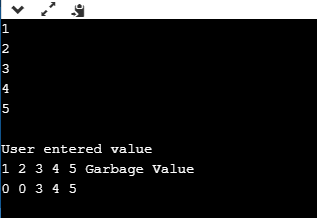
free() Return Value

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

1. #include <iostream>
2. #include <cstdlib>
3. **using** **namespace** std;
5. **int** main()
6. {
7. **int** \*ptr;
8. ptr = (**int**\*) malloc(5\***sizeof**(**int**));
9. cout << "Enter 5 integer" << endl;
11. **for** (**int** i=0; i<5; i++)
12. {
13. // \*(ptr+i) can be replaced by ptr[i]
14. cin >>ptr[i];
15. }
16. cout << endl << "User entered value"<< endl;
18. **for** (**int** i=0; i<5; i++)
19. {
20. cout <<\*(ptr+i)  << " ";
21. }
22. free(ptr);
24. /\* prints a garbage value after ptr is free \*/
25. cout << "Garbage Value" << endl;
27. **for** (**int** i=0; i<5; i++)
28. {
29. cout << \*(ptr+i)<< " ";
30. }
31. **return** 0;
32. }

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**



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);

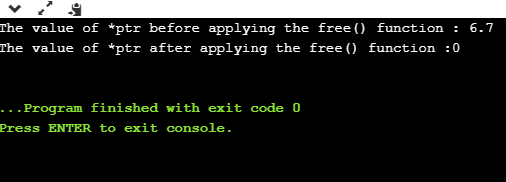
std::cout << "The value of \*ptr after applying the free() function :" <<\*ptr<< std::endl;

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:



#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

free vs delete in C++

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.

1. #include <iostream>
2. #include <cstdlib>
3. **using** **namespace** std;
5. **int** main()
6. {
7. **int** \*ptr;
8. ptr=**new** **int**;
9. \*ptr=68;
10. std::cout << "The value of p is : " <<\*ptr<< std::endl;
11. **delete** ptr;
12. std::cout <<"The value after delete is : "  <<\*ptr<< std::endl;
13. **return** 0;
14. }

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

free vs delete in C++

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

free vs delete in C++

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.