

# new and delete Operators in C++ For Dynamic Memory

Last Updated : 15 May, 2025



In C++, when a variable is declared, the compiler automatically reserves memory for it based on its data type. This memory is allocated in the program's **stack memory** at **compilation** of the program. Once allocated, it cannot be deleted or changed in size. However, C++ offers manual low-level memory management through dynamic memory allocation.

## What is Dynamic Memory Allocation?

**Dynamic memory allocation** is the process of **allocating** memory at the **runtime** of a program. It allows programmers to reserve some memory during the program's execution, use it as required and then free it to use it for some other purpose. **This memory is allocated in the Heap memory** of the program instead of the stack memory. It is very useful in cases like:

- When you are not sure about the size of the array you need.
- Implementing data structures such as linked list, trees, etc.
- In complex programs that require efficient memory management.

Dynamic memory allocation in C++ and deallocation is achieved by using two specialized operators: **new** and **delete**.

## new Operator

The **new** operator requests for the allocation of the block of memory of the given size of type on the **Free Store** (name for the part of heap memory available for new operator). If sufficient memory is available, a new operator initializes the memory to the default value according to its type and returns the address to this newly allocated memory.

## Syntax

```
new data_type;
```



In the above statement, a memory block that can store a single value of given **data\_type** is reserved in the heap and the address is returned. This address should be stored in the pointer variable of the same type.

### Example:

```
int *nptr = new int;
```

We allocated the memory for a single integer using new and stored its address in the integer pointer **nptr**. We can also initialize the allocated memory by providing an initial value:

```
int *nptr = new int(10);
```

### Example:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5
6      // Declared a pointer to store
7      // the address of the allocated memory
8      int *nptr;
9
10     // Allocate and initialize memory
11     nptr = new int(6);
12
13     // Print the value
14     cout << *nptr << endl;
15
16     // Print the address of memory
17     // block
18     cout << nptr;
19     return 0;
20 }
```

### Output

```
6
```

## Allocate Block of Memory (Array)

A **new** operator is also used to dynamically allocate a block (an [array](#)) of memory of given data type as shown below:

```
new data_type[n];
```

This statement dynamically allocates memory for **n** elements of given **data\_type**. Arrays can also be initialized during allocation.

### Example:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5
6      // Declared a pointer to store
7      // the address of the allocated memory
8      int *nptr;
9
10     // Allocate and initialize array of
11     // integer with 5 elements
12     nptr = new int[5]{1, 2, 3, 4, 5};
13
14     // Print array
15     for (int i = 0; i < 5; i++)
16         cout << nptr[i] << " ";
17     return 0;
18 }
```

### Output

```
1 2 3 4 5
```

What if enough memory is not available during runtime?

If enough memory is not available in the heap to allocate, the new request indicates failure by throwing an exception of type `std::bad_alloc`, unless "nothrow" is used with the new operator, in which case it returns a `nullptr` pointer. Therefore, it may be a good idea to check for the pointer variable produced by the new before using its program.

```
int *p = new (nothrow) int;
if (!p) {
    cout << "Memory allocation failed\n";
}
```

## delete Operator

In C++, `delete` operator is used to release dynamically allocated memory. It deallocates memory that was previously allocated with new.

### Syntax

```
delete ptr;
```

where, **ptr** is the pointer to the dynamically allocated memory.

To free the dynamically allocated array pointed by pointer variable, use the following form of delete:

```
delete[] arr;
```

### Example:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int *ptr = NULL;
6
7      // Request memory for integer variable
8      // using new operator
9      ptr = new int(10);
10     if (!ptr) {
11         cout << "allocation of memory failed";
12         exit(0);
13     }
```

```

13     }
14
15     cout << "Value of *p: " << *ptr << endl;
16
17     // Free the value once it is used
18     delete ptr;
19
20     // Allocate an array
21     ptr = new int[3];
22     ptr[2] = 11;
23     ptr[1] = 22;
24     ptr[0] = 33;
25     cout << "Array: ";
26     for (int i = 0; i < 3; i++)
27         cout << ptr[i] << " ";
28
29     // Deallocate when done
30     delete[] ptr;
31
32     return 0;
33 }

```

## Output

```

Value of *p: 10
Array: 33 22 11

```

## Errors Associated with Dynamic Memory

As powerful as dynamic memory allocation is it is also prone to one of the worst errors in C++. Major ones are:

### Memory Leaks

Memory leak is a situation where the memory allocated for a particular task remains allocated even after it is no longer needed. Moreover, if the address to the memory is lost, then it will remain allocated till the program runs.

**Solution:** Use smart pointers whenever possible. They automatically deallocate when goes out of scope.

## Dangling Pointers

Dangling pointers are created when the memory pointed by the pointer is accessed after it is deallocated, leading to undefined behaviour (crashes, garbage data, etc.).

**Solution:** Initialize pointers with nullptr and assign nullptr again when deallocated.

## Double Deletion

When delete is called on the same memory twice, leading to crash or corrupted program.

**Solution:** assign nullptr to the memory pointer when deallocated.

## Mixing new/delete with malloc()/free()


C++ supports the C style dynamic memory allocation using malloc(), calloc(), free(), etc. But these functions are not compatible. It means that we cannot allocate memory using new and delete it using free(). Same for malloc() and delete.

## Placement new

Placement new is a variant of **new** operator. Normal **new** operator both allocates memory and constructs an object in that memory. On the other hand, the placement new separates these actions. It allows the programmer to pass a pre-allocated memory block and construct an object in that specific memory.

Dynamic Memory Allocation

[Visit Course ↗](#)

 Comment

More info ▼

Advertise with us

**Next Article** >

new vs malloc() and free() vs delete in  
C++

## Similar Reads

# C++ Programming Language

C++ is a computer programming language developed by Bjarne Stroustrup as an extension of the C language. It is known for its fast speed, low level memory management and is often taught as first...

🕒 5 min read

---

**C++ Overview** ▼

---

**C++ Basics** ▼

---

**C++ Variables and Constants** ▼

---

**C++ Data Types and Literals** ▼

---

**C++ Operators** ▼

---

**C++ Input/Output** ▼

---

**C++ Control Statements** ▼

---

**C++ Functions** ▼

---

**C++ Pointers and References** ▼

---