# C++ Dynamic Memory

A good understanding of how dynamic memory really works in C++ is essential to becoming a good C++ programmer. Memory in your C++ program is divided into two parts −

- **The stack** − All variables declared inside the function will take up memory from the stack.
- **The heap** − This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory that was previously allocated by new operator.

## new and delete Operators

There is following generic syntax to use **new** operator to allocate memory dynamically for any data-type.

```
new data-type;
```

Here, **data-type** could be any built-in data type including an array or any user defined data types include class or structure. Let us start with built-in data types. For example we can define a pointer to type double and then request that the memory be allocated at execution time. We can do this using the **new** operator with the following statements −

```
double* pvalue  = NULL; // Pointer initialized with null
pvalue  = new double;   // Request memory for the variable
```

The memory may not have been allocated successfully, if the free store had been used up. So it is good practice to check if new operator is returning NULL pointer and take appropriate action as below −

```
double* pvalue  = NULL;
if( !(pvalue  = new double )) {
```

```
   cout << "Error: out of memory." <<endl;
   exit(1);
}
```

The **malloc()** function from C, still exists in C++, but it is recommended to avoid using malloc() function. The main advantage of new over malloc() is that new doesn't just allocate memory, it constructs objects which is prime purpose of C++.

At any point, when you feel a variable that has been dynamically allocated is not anymore required, you can free up the memory that it occupies in the free store with the 'delete' operator as follows −

```
delete pvalue;        // Release memory pointed to by pvalue
```

Let us put above concepts and form the following example to show how 'new' and 'delete' work −

Live Demo

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

int main () {
   double* pvalue  = NULL; // Pointer initialized with null
   pvalue  = new double;   // Request memory for the variable

   *pvalue = 29494.99;     // Store value at allocated address
   cout << "Value of pvalue : " << *pvalue << endl;

   delete pvalue;         // free up the memory.

   return 0;
}
```

If we compile and run above code, this would produce the following result −

```
Value of pvalue : 29495
```

## Dynamic Memory Allocation for Arrays

Consider you want to allocate memory for an array of characters, i.e., string of 20 characters. Using the same syntax what we have used above we can allocate memory dynamically as shown below.

```
char* pvalue  = NULL;         // Pointer initialized with null
pvalue  = new char[20];       // Request memory for the variable
```

To remove the array that we have just created the statement would look like this −

```
delete [] pvalue;            // Delete array pointed to by pvalue
```

Following the similar generic syntax of new operator, you can allocate for a multi-dimensional array as follows −

```
double** pvalue  = NULL;     // Pointer initialized with null
pvalue  = new double [3][4]; // Allocate memory for a 3x4 array
```

However, the syntax to release the memory for multi-dimensional array will still remain same as above −

```
delete [] pvalue;            // Delete array pointed to by pvalue
```

## Dynamic Memory Allocation for Objects

Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept −

Live Demo

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

class Box {
   public:
      Box() {
         cout << "Constructor called!" <<endl;
      }
      ~Box() {
         cout << "Destructor called!" <<endl;
      }
};
int main() {
   Box* myBoxArray = new Box[4];
   delete [] myBoxArray; // Delete array

   return 0;
}
```

If you were to allocate an array of four Box objects, the Simple constructor would be called four times and similarly while deleting these objects, destructor will also be called same number of times.

If we compile and run above code, this would produce the following result −

```
Constructor called!
Constructor called!
Constructor called!
Constructor called!
Destructor called!
Destructor called!
Destructor called!
Destructor called!
```