**Dynamic Memory Allocation in C++**

Dynamic memory allocation in C/C++ refers to performing memory allocation manually by programmer. Dynamically allocated memory is allocated on **Heap** and non-static and local variables get memory allocated on **Stack.**

**What are applications?**

* One use of dynamically allocated memory is to allocate memory of variable size which is not possible with compiler allocated memory except [variable length arrays](https://www.geeksforgeeks.org/variable-length-arrays-in-c-and-c/).
* The most important use is flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need and whenever we don’t need anymore. There are many cases where this flexibility helps. Examples of such cases are [Linked List](https://www.geeksforgeeks.org/data-structures/linked-list/), [Tree](https://www.geeksforgeeks.org/binary-tree-2/), etc.

**How is memory allocated/deallocated in C++?**

C uses [malloc() and calloc()](https://www.geeksforgeeks.org/calloc-versus-malloc/) function to allocate memory dynamically at run time and uses free() function to free dynamically allocated memory. C++ supports these functions and also has two operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way.

**new operator**

The new operator denotes a request for memory allocation on the Heap. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

* **Syntax to use new operator**: To allocate memory of any data type, the syntax is:
* pointer-variable = **new** data-type;

Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user defined data types including structure and class.  
Example:

// Pointer initialized with NULL

// Then request memory for the variable

int \*p = NULL;

p = new int;

OR

// Combine declaration of pointer

// and their assignment

int \*p = new int;

* **Initialize memory:** We can also initialize the memory using new operator:

pointer-variable = **new** data-type(value);

**Example:**

int \*p = new int(25);

float \*q = new float(75.25);

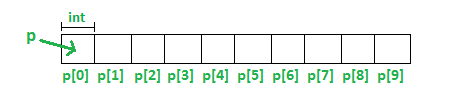
* **Allocate block of memory:** new operator is also used to allocate a block(an array) of memory of type *data-type*.
* pointer-variable = **new** data-type[size];

where size(a variable) specifies the number of elements in an array.

Example:

int \*p = new int[10]

Dynamically allocates memory for 10 integers continuously of type int and returns pointer to the first element of the sequence, which is assigned to p(a pointer). p[0] refers to first element, p[1] refers to second element and so on.

[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/dynamic.png)

**Delete operator**

Since it is programmer’s responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.

**Syntax:**

// Release memory pointed by pointer-variable

**delete** pointer-variable;

Here, pointer-variable is the pointer that points to the data object created by *new*.  
Examples:

delete p;

delete q;

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

// Release block of memory

// pointed by pointer-variable

Delete [] pointer-variable;

Example:

// It will free the entire array

// pointed by p.

delete[] p;

Note: difference between **delete** and **delete[]** in C++

|  |  |
| --- | --- |
| #include <iostream>  usingnamespacestd;  int main(){  int \*p = newint [30];  p[0] = 1;  p[1] = 5;  cout<< p[0] <<endl<< p[1] <<endl<<endl;  delete p;  cout<< p[0] <<endl<< p[1] <<endl<<endl;  return 0;  } |  |

delete p will delete only p[0]. Other 29 entries will not be deleted and this called **memory leak.**  
delete[] p will delete whole array.

The reason why there are separate **delete** and **delete**[] operators is that **delete** calls one destructor whereas **delete**[] needs to look up the size of the array and call that many times the destructor function.

**1D Array**

#include <iostream>

#define N 10

// Dynamically Allocate Memory for 1D Array in C++

int main()

{

// dynamically allocate memory of size N

int\* A = new int[N];

// assign values to allocated memory

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

A[i] = i + 1;

// print the 1D array

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

std::cout<< A[i] << " "; // or \*(A + i)

// deallocate memory

delete[] A;

return 0;

}

**2D Array**

#include <iostream>

// M x N matrix

#define M 4

#define N 5

// Dynamically Allocate Memory for 2D Array in C++

int main()

{

// dynamically create array of pointers of size M

int\*\* A = new int\*[M];

// dynamically allocate memory of size N for each row

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

A[i] = new int[N];

// assign values to allocated memory

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

for (int j = 0; j < N; j++)

A[i][j] = rand() % 100;

// print the 2D array

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

{

for (int j = 0; j < N; j++)

std::cout<< A[i][j] << " ";

std::cout<<std::endl;

}

// deallocate memory using delete[] operator

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

delete[] A[i];

delete[] A;

return 0;

}

Note:

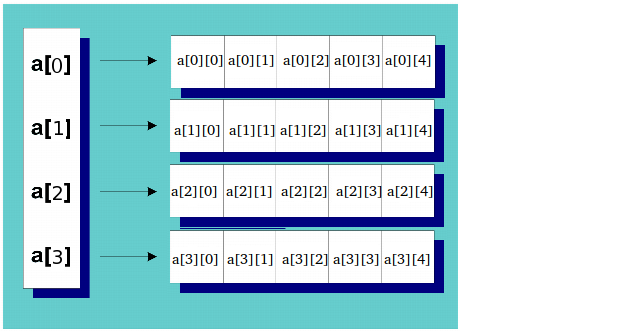
A dynamic 2D array is basically an array of *pointers to arrays*. You can initialize it using a loop, like this:

int\*\* a = new int\*[rowCount];

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

a[i] = new int [colCount];

The above, for colCount= 5 and rowCount = 4, would produce the following:



and then **clean up would be:**

for(int i =0; i <sizeY;++i){

delete[] ary[i];

}

delete[] ary;

**3D Array**

#include <iostream>

// X x Y x Z matrix

#define X 2

#define Y 3

#define Z 4

// Dynamically Allocate Memory for 3D Array in C++

int main()

{ int\*\*\* A = new int\*\*[X];

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

{

A[i] = new int\*[Y];

for (int j = 0; j < Y; j++)

A[i][j] = new int[Z];

}

// assign values to allocated memory

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

for (int j = 0; j < Y; j++)

for (int k = 0; k < Z; k++)

A[i][j][k] = rand() % 100;

// print the 3D array

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

{

for (int j = 0; j < Y; j++)

{

for (int k = 0; k < Z; k++)

std::cout<< A[i][j][k] << " ";

std::cout<<std::endl;

}

std::cout<<std::endl;

}

// deallocate memory

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

{

for (int j = 0; j < Y; j++)

delete[] A[i][j];

delete[] A[i];

}

delete[] A;

return 0;

}

**Following are the differences between malloc() and operator new.**:

1. **Calling Constructors:**new calls constructors, while malloc() does not. In fact primitive data types (char, int, float..etc) can also be initialized with new. For example, below program prints 10.

|  |
| --- |
| #include<iostream>   Using namespace std;   int main()  {     int\*n = new int(10); // initialization with new()     cout<< \*n;     getchar();     return0;  } |

1. **operators function:** new is an operator, while malloc() is a function.
2. **return type:** new returns exact data type, while malloc() returns void \*.
3. **Failure Condition:**On failure, malloc() returns NULL where as new Throws.
4. **Memory:** In case of new, memory is allocated from free store where as in malloc() memory allocation is done from heap.
5. **Overriding:** We are allowed to override new operator where as we cannot override the malloc() function legally.
6. **Size:** Required size of memory is calculated by compiler for new, where as we have to manually calculate size for malloc().

|  |  |
| --- | --- |
| **NEW** | **MALLOC** |
| calls constructor | Does not call constructors |
| It is an operator | It is a function |
| Returns exact data type | Returns void \* |
| on failure, Throws | On failure, returns NULL |
| Memory allocated from free store | Memory allocated from heap |
| can be overridden | cannot be overridden |
| size is calculated by compiler | size is calculated manually |