Data Structure Lab

Lab # 02

Dynamic Memory Allocation

Instructor: Muhammad Saood Sarwar

Email: saood.sarwar@nu.edu.pk

Course Code: CL2001

Department of Computer Science,

National University of Computer and Emerging Sciences FAST Peshawar Campus

Table of Contents

C++ Dynamic Memory **2**

new and delete operators in C++ for dynamic memory 3

new operator 3

delete operator 5

Dynamic Memory Allocation for Objects 8

Dynamic memory allocation in C++ for 1D and 2D **9**

1. Single Dimensional Array **9**

2. 2-Dimensional Array **10**

Using array of Pointers 10

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 in C++ for dynamic memory.

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 (Refer Memory Layout C Programs for details).
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.
- 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, Tree, etc.

How is it different from memory allocated to normal variables?
For normal variables like "int a", "char str[10]", etc, memory is automatically allocated and deallocated. For dynamically allocated memory like "int *p = new int[10]", it is programmers responsibility to deallocate memory when no longer needed. If programmer doesn't deallocate memory, it causes memory leak (memory is not deallocated until program terminates).
How is memory allocated/deallocated in C++?
C uses malloc() and calloc() 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.
This is all about new and delete operators.

new operator

The new operator denotes a request for memory allocation on the Free Store. 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:

```
// 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 for built-in data types using new operator. For custom data types a constructor is required (with the data-type as input) for initializing the value. Here's an example for the initialization of both data types :

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

Example 1

```
#include <iostream>
using namespace std;
int main(){
int* ptr = new int(25);
float* f_ptr = new float(75.25);
// Custom data type
struct cust
{
int p;
cust(int q){
 p = q;
}

};
cust* var = new cust(10);
cout<<*ptr<<endl;
```

```
cout<<*f_ptr<<endl;
cout<<var->p<<endl;
    return 0;
}
```

- **Allocate block of memory:** new operator is also used to allocate a block(an array) of memory of type data-type.

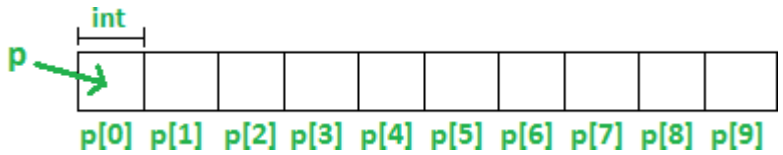
pointer-variable = new data-type[size];

- 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.



Normal Array Declaration vs Using new

There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, normal arrays are deallocated by compiler (If array is local, then deallocated when function returns or completes). However, dynamically allocated arrays always remain there until either they are deallocated by programmer or program terminates.

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 NULL pointer (scroll to section "Exception handling of new operator" in this article). Therefore, it may be good idea to check for the pointer variable produced by new before using it program.

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

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;
```

Example 2

```
// C++ program to illustrate dynamic allocation

// and deallocation of memory using new and delete

#include <iostream>

using namespace std;

int main()

{

// Pointer initialization to null

int* p = NULL;

// Request memory for the variable using new operator

p = new (nothrow) int;
```

```
if (!p)
cout << "allocation of memory failed\n";
else
{
// Store value at allocated address
*p = 29;
cout << "Value of p: " << *p << endl;
}
// Request block of memory using new operator
float* r = new float(75.25);
cout << "Value of r: " << *r << endl;
// Request block of memory of size n
int n = 5;
int* arr = new int[n];
if (!arr)
   cout << "allocation of memory failed\n";
else
{
   for (int i = 0; i < n; i++)
       arr[i] = i + 1;
   cout << "Value store in block of memory using index: ";
   for (int i = 0; i < n; i++)
    cout << arr[i] << " ";
}
// freed the allocated memory
delete p;
delete r;
```

```
// freed the block of allocated memory

delete[] arr;

return 0;

}
```

Output:

```
Value of p: 29

Value of r: 75.25

Value store in block of memory using index: 1 2 3 4 5
```

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 −

Example 3

```
#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!
```

Dynamic memory allocation in C++ for 1D and 2D

This post will discuss dynamic memory allocation in C++ for multidimensional arrays.

1. Single Dimensional Array

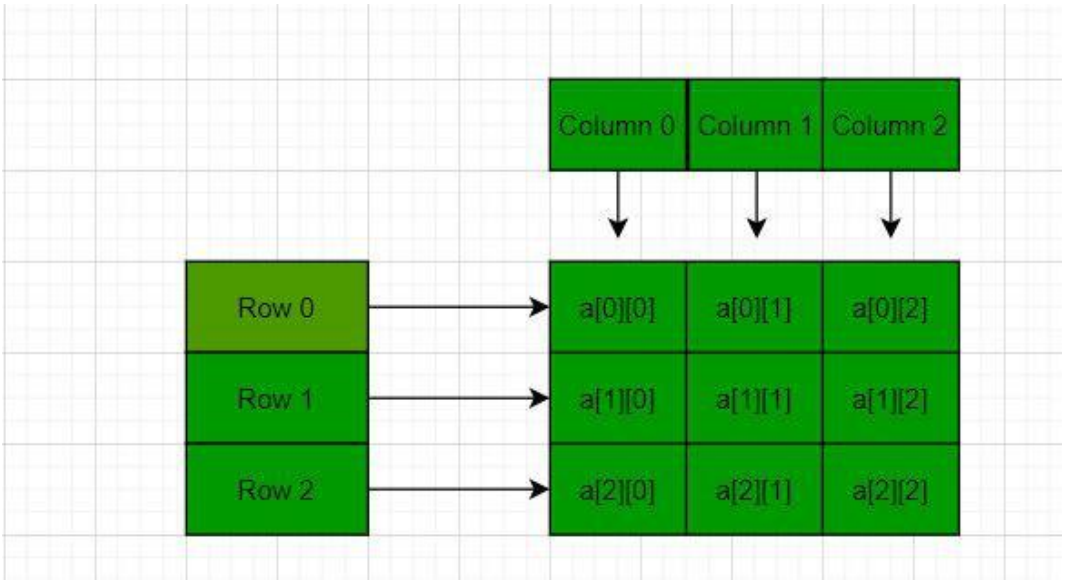
Example 4

```
#include <iostream>
using namespace std;
#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 the allocated memory
    for (int i = 0; i < N; i++)
    {
        A[i] = i + 1;
    }
    // print the 1D array
    for (int i = 0; i < N; i++)
    {
        cout << A[i] << " ";   // or *(A + i)
    }
    // deallocate memory
    delete[] A;
    return 0;
}
```

2. 2-Dimensional Array

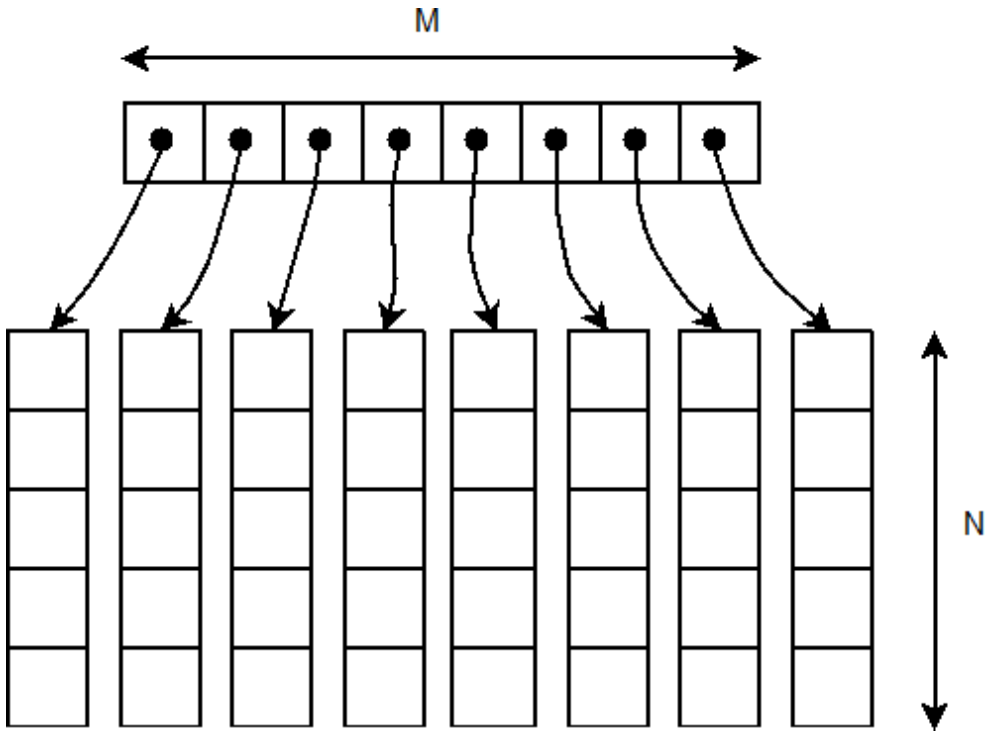
In C/C++, multidimensional arrays in simple words as an array of arrays. Data in multidimensional arrays are stored in tabular form (in row major order).

Below is the diagrammatic representation of 2D arrays:



Using array of Pointers

We can dynamically create an array of pointers of size `M` and then dynamically allocate memory of size `N` for each row, as shown below:



Example 5

```
#include <iostream>
using namespace std;
#define M 2
#define N 3
// Dynamic Memory Allocation in C++ for 2D Array
int main()
{
// dynamically create an 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 the 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++)
   {
       cout << A[i][j] << " ";
   }
cout << endl;
}
// deallocate memory using the delete[] operator
for (int i = 0; i < M; i++)
{
```

```
delete[] A[i];
}
delete[] A;
return 0;
}
```