Dynamic Memory Allocation 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.

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:**

  ```
  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:
  - ***Syntax:***

    ```
    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*.
  - ***Syntax:***

    ```
    pointer-variable = new data-type[size]:
    ```

    Where size(a variable) specifies the number of elements in an array.
  - ***Example:***

    ```
    int *p = new int[10]:
    ```

- **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. Therefore, it may be good idea to check for the pointer variable produced by new before using it program.
  - ***Syntax:***

```
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:***

```
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*:
  - ***Syntax:***

```
// Release block of memory, pointed by pointer-variable
delete[] pointer-variable:
```

  - ***Example:***

```
// It will free the entire array pointed by p
delete[] p:
```

- **malloc() vs 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.
  2. **Operator vs Function:** new is an operator, while malloc() is a function.
  3. **Return Type:** new returns exact data type, while malloc() returns void *.
  4. **Failure Condition:** On failure, malloc() returns NULL where as new throws bad_alloc exception.
  5. **Memory:** In case of new, memory is allocated from free store where as in malloc() memory allocation is done from heap.
  6. **Size:** Required size of memory is calculated by compiler for new, where as we have to manually calculate size for malloc().
  7. **Buffer Size:** malloc() allows to change the size of buffer using realloc() while new doesn't.

- **free() vs delete:**
  1. The delete is an operator that de-allocates the memory dynamically while the free() is a function that destroys the memory at the runtime.
  2. 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.
  3. 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.
  4. The delete() operator is faster than the free() function.

- **The following summarize a C++ program's major distinct memory areas.**

Memory Area	Characteristics and Object Lifetimes
Const Data	The const data area stores string literals and other data whose values are known at compile time. No objects of class type can exist in this area. All data in this area is available during the entire lifetime of the program. Further, all of this data is read-only, and the results of trying to modify it are undefined. This is in part because even the underlying storage format is subject to arbitrary optimization by the implementation. For example, a particular compiler may store string literals in overlapping objects if it wants to.
Stack	The stack stores automatic variables. Typically allocation is much faster than for dynamic storage (heap or free store) because a memory allocation involves only pointer increment rather than more complex management. Objects are constructed immediately after memory is

	allocated and destroyed immediately before memory is deallocated, so there is no opportunity for programmers to directly manipulate allocated but uninitialized stack space (barring willful tampering using explicit dtors and placement new).
Free Store	The free store is one of the two dynamic memory areas, allocated/freed by new/delete. Object lifetime can be less than the time the storage is allocated; that is, free store objects can have memory allocated without being immediately initialized, and can be destroyed without the memory being immediately deallocated. During the period when the storage is allocated but outside the object's lifetime, the storage may be accessed and manipulated through a void* but none of the proto-object's non-static members or member functions may be accessed, have their addresses taken, or be otherwise manipulated.
Heap	The heap is the other dynamic memory area, allocated/freed by malloc/free and their variants. Note that while the default global new and delete might be implemented in terms of malloc and free by a particular compiler, the heap is not the same as free store and memory allocated in one area cannot be safely deallocated in the other. Memory allocated from the heap can be used for objects of class type by placement-new construction and explicit destruction. If so used, the notes about free store object lifetime apply similarly here.
Global/Static	Global or static variables and objects have their storage allocated at program startup, but may not be initialized until after the program has begun executing. For instance, a static variable in a function is initialized only the first-time program execution passes through its definition. The order of initialization of global variables across translation units is not defined, and special care is needed to manage dependencies between global objects (including class statics). As always, uninitialized proto-objects' storage may be accessed and manipulated through a void* but no non-static members or member functions may be used or referenced outside the object's actual lifetime.