Full Class Array (with Big Three)

```
template <typename T>
class ABQ {
    pri vate:
   T* data;
    size_t curren t_size;
    size_t max_ca pacity;
    size_t front;
    size_t back;
    float scale_ factor;
    public:
ABQ() {
   this -> max_ca pacity = 1;
   this -> curren t_size = 0;
   this -> front = 0;
   this -> back = 0;
    thi s-> sca le_ factor =
2.0f;
   data = new T[max_ cap acity];
}
ABQ(int capacity) {
   this -> max_ca pacity =
capacity;
   this -> curren t_size = 0;
   this -> front = 0;
   this -> back = 0;
    thi s-> sca le_ factor =
2.0f;
   data = new T[max_ cap acity];
}
ABQ(const ABQ &d) {
   this -> max_ca pacity =
d.max_ cap acity;
   this -> curren t_size =
d.curr ent _size;
   this -> front = d.front;
   this -> back = d.back;
    thi s-> sca le_ factor =
d.scal e_f actor;
   data = new T[max_ cap acity];
   for (size_t i = 0; i <
curren t_size; i++) {
        data[i] = d.data[i];
```

Full Class Array (with Big Three) (cont)

```
> }
}
ABQ &operator=(const ABQ &d) {
 if (this != &d) {
   delete [] data;
   this -> max_capacity = d.max_capacity;
   this -> current_size = d.current_size;
   this -> front = d.front;
   this -> back = d.back;
   this->scale_factor = d.scale_factor;
   data = new T[max_capacity];
   for (size_t i = 0; i < current_size; i++) {
    data[i] = d.data[i];
   }
 }
 return *this;
}
~ABQ() {
 delete [] data;
}
```

Module 1

```
#pragma once // prevents
including multiple times
#include “funct ions.h” // gets
functions & classes from file
#include <io str eam> //
includes official string
directory
FUNCTIONS
int addNum(int x, int y); //
prototype
int addNum(int a, int b){ //
definition
      return a + b;
}
```

Pointers

```
int x = 16;
int xPtr = &x; // Creates
pointer int xPtr2 = &x; // Also
works
int* ptr = nullptr; // Points to
nothing
*xPtr = 10; // Changes x
// References must be initia -
lized
int& xRef = x; // Creates
reference to x int &xRef2 = x;
// Also works
// xRef and x have same memory
address
xRef = 10; // Changes x and xRef
// Changes variable passed in
through pointer
void passBy Poi nte r(int* x){
     *x = 30;
}
// Changes variable passed in
through reference
void passBy Ref (in t& x){
     x = 40;
}
```

Constant

```
Foo::Foo(const Foo&); // passes
in the parameter as a const
const int Foo::g etD ata(); //
returns a const int
int Foo::g etC oun t(int index)
const; // makes function const
int someVal = 50;
// Cannot modify the value by
derefe rencing a pointer
const int* pointer1 = &s omeVal;
// Cannot change the address the
pointer is pointing to
int* const pointer2 = &s omeVal;
const int* const pointer3 = &s -
omeVal; // Can't change anything
```

C Style

In C, the end of a char* C-style string is defined by the null terminator ('\0'). This special character marks the end of the string in memory, allowing functions like strlen(), strcpy(), and strcmp() to know where the string ends. Without the null terminator, these functions would continue reading memory beyond the string, causing errors. For example, the string "Hello" in memory is stored as ['H', 'e', 'l', 'l', 'o', '\0']. The most common way to define a null-terminated string in C is by using a string literal, such as char str[] = "Hello";, which automatically includes the null terminator. Defining a string manually without the null terminator, like char str[] = {'H', 'e', 'l', 'l', 'o'};, is incorrect because it doesn't mark the end of the string. Another valid but less common way is char str[] = {'H', 'e', 'l', 'l', 'o', '\0'};. However, using string literals is the preferred and more concise method..

Operator Overloading

```
DynamicArray operator+(const
DynamicArray& other) {
       int newSize = max(size,
other.s ize);
        Dyn ami cArray temp(n -
ewS ize); // Create new array
with size based on the max size
       for (int i = 0; i < size
&& i < other.s ize; ++i) {
                tem p.a rr[i] =
arr[i] + other.a rr[i];
       }
        return temp;
}
Dynami cAr ray & operat or+ =
(const Dynami cAr ray & other) {
```

Operator Overloading (cont)

```
>    int newSize = max(size, other.size);
   if (newSize > capacity) {
      resize(newSize); // Resize if necessary
   }
   for (int i = 0; i < other.size; ++i) {
      arr[i] += other.arr[i];
   }
   size = newSize;
   return *this; // Return current object for
chaining
}
DynamicArray operator-(const DynamicAr-
ray& other) {
   int newSize = max(size, other.size);
   DynamicArray temp(newSize); // Create
new array with size based on the max size
   for (int i = 0; i < size && i < other.size; ++i)
{
      temp.arr[i] = arr[i] - other.arr[i];
   }
   return temp;
}
DynamicArray& operator-=(const Dynami-
cArray& other) {
   int newSize = max(size, other.size);
   if (newSize > capacity) {
      resize(newSize); // Resize if necessary
   }
   for (int i = 0; i < other.size; ++i) {
      arr[i] -= other.arr[i];
   }
   size = newSize;
   return *this; // Return current object for
chaining
}
bool operator==(const DynamicArray&
other) const {
```

Operator Overloading (cont)

```
>    if (size != other.size) {
      return false;
   }
   for (int i = 0; i < size; ++i) {
      if (arr[i] != other.arr[i]) {
         return false;
      }
   }
   return true;
}
bool operator!=(const DynamicArray& other)
const {
   return !(*this == other); // Using ==
operator
}
```