CISS245: Advanced Programming
Assignment 10

Name: _____

Objectives

- Declare objects
- Access member variables of an object
- Write member functions/operators
- Work with classes containing pointer members
- Overwrite default methods (constructors, destructors)
- Overload operators

Q1. The objective is to write a class, `IntDynArr`, that models a dynamic array of integers. In particular, the class to be implemented uses system resources (i.e. memory) and hence several default operators (example: the destructor, copy constructor and `operator=`) must be overwritten.

You will need to write three files:

- `testIntDynArr.cpp`: This contains a program to test the `IntDynArr` class and it's supporting functions and operators.
- `IntDynArr.h`: This is the header function containing the `IntDynArr` class.
- `IntDynArr.cpp`: This file contains the definitions for the methods in `IntDynArr.h`.

As before all methods/functions implemented should be tested carefully in the test code.

Read the following carefully before you dive straight into coding. As before, you should minimize code duplication, provide reasonable access control (i.e., `const` a reference parameter whenever possible), code with proper indentation, etc.

Such a class is extremely important because in the real world, we want to work with arrays with different sizes during runtime. In fact it's so important that C++ actually has a library call the Standard Template Library (STL) that includes a class similar to the class in this assignment.

With this class written, you can do the following easily:

```
// Work with x which simulates an array of 5 integers
IntDynArr x(5);
x.resize(5);
for (int i = 0; i < 5; i++)
{
    x[i] = 2 * i + 1;
}
std::cout << x << std::endl; // prints [1, 3, 5, 7, 9]

x.insert(0, -100); // put -100 at index 0
std::cout << x << std::endl; // prints [-100, 1, 3, 5, 7, 9]
// Note that now x simulates an array of size 6, not 5.
// Therefore your array class actually allows x to change its
// size auto-magically!

x.remove(3); // remove value at index 3
```

```
std::cout << x << std::endl; // prints [-100, 1, 3, 7, 9]
// Note that now x simulates an array of size 5.

x.remove(3); // remove value at index 3
std::cout << x << std::endl; // prints [-100, 1, 3, 9]
// Note that now x simulates an array of size 4.
```

Other methods and operators are described more fully in the next section.

Once this class is completed, you can easily modify it for different values such as a dynamic array of strings, of doubles, of alien spaceships, etc.

Requirements

You know that C/C++ arrays have constant sizes if the arrays are in your local blocks scope. One way to "make" an array dynamic is to use pointers to point to arrays in the heap. Hence it makes sense to create a class that models a "dynamic" array and hide all the complexities in this class.

The private members of `IntDynArr` includes

- `size` (an `int` – you can name it `size_` if you wish),
- `capacity` (an `int` – you can name it `capacity_` if you wish), and
- `x` (an `int *` – you can name it `x_` if you wish)

What is the purpose of the `capacity` instance variable? (I've talked about this in CISS240. Also, refer to the notes in CISS245.)

Suppose you construct an `IntDynArr` object that models an array with `size` 100. At a later point in time, you might want the object to model an array of `size` 90 instead. You can for instance allocate new memory of 90 integers for this object to refer, copy the contents of the old array to the new, and deallocate the original 100 integers. But the problem is, suppose you later want to model an array of 100 integers again. You have to do the memory deallocation and allocation again. A better scheme in this case is to allocate memory for 100 integers and set the `size` to 100. When you need the array size to be 90, just set the `size` to 90; you still have 100 integers available, it's just that 10 of them should not be accessed. When you need 100 integers again, just set the `size` to 100. See notes below for details on `size` and `capacity`.

Recall that since each `IntDynArr` object has a pointer member that points to an array of integers (i.e. it uses system resource that is not automatically released), you need to overwrite the

- Default destructor `~IntDynArr()` to release the memory x is pointing to
- Default copy constructor `IntDynArr(const IntDynArr &)`
- Default `IntDynArr & operator=(const IntDynArr &)`

The reason for overwriting the above default methods are already mentioned in class.

Here are the methods/features of the class:

1. `IntDynArr(int capacity0 = 5)`: Constructs an object so that `size` is set to `0`, `capacity` is set to `capacity0`, and `x` points to an integer array in the heap with `capacity0` integers.
2. `IntDynArr(int size0, int a[])`: Constructs an object so that `size` and `capacity` are both set to `size0`, and `x` points to an integer array of size `size0`. The values `a[0]`, ..., `a[size0 - 1]` are copied to the array `x` points to.

3. `IntDynArr(const IntDynArr &)`: This is the obvious copy constructor.
4. `~IntDynArr()`: The destructor must release all resources held by an object.
5. `int get_size() const`: returns the value of the `size`.
6. `int get_capacity() const`: returns the value of the `capacity`.
7. `bool operator==(const IntDynArr & arr) const`: Returns `true` if the array object `*this` and `arr` are the same in the sense that their sizes and "contents" are the same. Here, "contents" mean the values in the array from index 0 to `size - 1`.
8. `bool operator!=(const IntDynArr & arr) const`: The opposite of `operator==`.
9. `IntDynArr & operator=(const IntDynArr & arr)`: After assignment, `*this==arr` would return `true`. See notes below.
10. `std::ostream & operator<<(std::ostream &, const IntDynArr & a)`: Prints the contents of the array `a.x` is pointing to. Integers are separated by spaces. For instance if `a.x` points to the array 4, 5, 8, 9,2 and `a.size` is 3, then "[4, 5, 8]" is printed (without the quotes of course). Note that this is a function outside the class, i.e., it's a nonmember function.
11. `std::istream & operator>>(std::istream &, IntDynArr & a)`: Read in a list of integers from the keyboard until you see a -1 and fill in the dynamic array with the integers you just read. You must clear the existing values in the array and adjust its size and capacity as appropriate. As you should know by now, this is also a nonmember function.
12. `void resize(const int size0)`: This method "resizes" the array `x` is pointing to. This is what it does: Suppose `a` is an `IntDynArr` object that models an array of `size` 10. After calling `a.resize(15)`, the `a` models an array of size 15. The contents of the original array of 10 integers are copied over to the new array of size 15 starting at index 0. This means that 5 elements of the new array can contain arbitrary values. Note that there should not be any memory leaks after the call. For the above example, the array of 10 integers `a.x` was pointing to must be released back to the heap. See notes below for details.
13. `int operator[](int i) const` and `int & operator[](int i)`: Note that there are two operators here. They returns `x[i]` either as an integer value or an integer reference. The first allows us to do this:

```
IntDynArr arr(5);
std::cout << arr[0] << std::endl;  // arr[0] is arr.operator[](0)
```

Now you might wonder why are we need a version `operator[]` that returns a reference (`int &`) instead of just an integer. If `operator[]` returns an integer, then the return value is an integer value and in that case

```
std::cout << arr[0] << std::endl;  // arr[0] is arr.operator[](0)
```

works. But

```
arr[0] = 3;
```

does not work. If `arr.x[0]` is 5 and we returned an `int` (instead of an integer reference), then "`arr[0] = 3`" would be the same as saying "`5 = 3;`" which is of course nonsense. However if `operator[]` returns a reference, in that case what is returned in not an integer but a reference to `arr.x[0]`, an integer "variable". (Refer to your lecture notes on references.) Therefore for the statement

```
arr[0] = 3;
```

the second `operator[]` (the one that returns a reference) is called. This was already mentioned in class several times when I talked about returning references.

14. `IntDynArr & operator+=(const IntDynArr & a)`: This operator will "append" (or concatenate) the contents of `a` to the object invoking this operator. For instance if object `x` models the array 1, 2, 3, and object `a` models the array 7, 8, 9, then `x += a` will modify `x` so that it models the array 1, 2, 3, 7, 8, 9.
15. `IntDynArr operator+(const IntDynArr & a) const`: This operator will return a copy of the object which is the concatenation of the contents of the object invoking the call with the contents of `a`. For instance of object `x` models the array 1, 2, 3, and object `a` models the array 7, 8, 9, then `x + a` will return an object that models the array 1, 2, 3, 7, 8, 9.
16. `IntDynArr & insert(int index, int val)`: This will modify the object invoking this method by inserting value val at index `index`. For instance if the object `x` models the array 1, 2, 3, after the method call `x.insert(1, 9)`, `x` will model the array 1, 9, 2, 3.
17. `IntDynArr & remove(int index)`: This will modify the object invoking this method by removing the value at index `index`. For instance if the object `x` models the array 1, 2, 3, after the method call `x.remove(1)`, `x` will model the array 1, 3.
18. `IntDynArr subarray(int index, int length = -1) const`: This will return an
    `IntDynArr` object which is "part of" the object invoking the method. For instance if the object `x` models the array 90, 91, 92, 93, 94, 95, 96, then `x.subarray(3, 2)` will return an `IntDynArr` object that models the array 93, 94. If the length argument is not specified, then all the values from the specified index to the end of object are used in creating the new object. For instance, with the above object `x`, calling `x.subarray(3)` will return an `IntDynArr` object that models the array 93, 94, 95, 96.
19. `void print() const`: This is the `std::ostream & operator<<(std::ostream &, IntDynArr & a)` except that it prints the `size` and `capacity` as well. It also prints a newline. This can be used for debugging purposes. For instance if `a.x` points to the array 4, 5, 8, 9, 2 and `a.size` is 3, then "`[4, 5, 8], size:3, capacity:5\n!`" is printed (without the quotes of course).

Note that you will see many actions taken by methods are similar. For instance

arrays are copied in both the copy constructor and `operator=`. You must avoid code duplication. You can create "helper" functions used by methods in your class. These can be either functions or methods. For example:

```
// C.h
void helper(...);
class C
{
    ...
};

// C.h
class C
{
    ...
    void helper(...);
    ...
};
```

Note that if declared as a method, you can "hide" the helper method from outsiders by placing it under the `private` section of the class:

```
// C.h
class C
{
    ...
private:
    ...
    void helper(...);
    ...
};
```

Error on Allocating Memory

It is a good practice (and you must follow it) to check your pointer after a memory allocation. Failure of memory allocation can be checked by comparing the pointer's value. If memory is not allocated, the pointer is set to `NULL`. The following is an example:

```
int * p = new int;
if (p == NULL)
{
    std::cout << "ERROR!!! You ran out of member!!!"
}
else // memory is allocated
{
    ...
}
```

If a pointer is not allocated memory after a new operator call, your program must print the string "ERROR: memory allocation returns NULL".

This is the only action you have to take when there is a memory allocation error. In a real software, it's likely that the program will print a message and then completely halt the program.

(Actually there are two possible ways to check for memory allocation errors and both should be used if possible. We won't be able to talk about the other method because it involves the concept of "exceptions" which we will be talk about later.)

`operator=` and copying

[This section is included for your benefit. We already talked about the problems of direct member-wise copying for pointer members in class.]

The following notes involves some form of copying between objects in the context where pointers are involved. Suppose we first look at a very simple situation:

```
class Dummy
{
private:
    int x;
    double y;
    char z;
};
```

If you execute the following code:

```
Dummy a, b;
b = a;  // b invokes operator= with a as argument
Dummy c = a;    // c invokes copy constructor with a as argument
```

In both cases, the values of instance variables of `a` are copied to corresponding instance variables of `b` and `c`. In other words, `a.x` is copied over to `b.x`, etc.

This is not always appropriate. For instance in the case where an instance variable points to an array. The following discussion on copying the contents from an object to another there a pointer to an array is involved is relevant to any copying behavior in an object, including `operator=` and the copy constructor.

Suppose `a` is an object where the `x` member is a pointer to an array of 4 integers in the heap:

(The following has nothing to do with the values of `size` and `capacity` so I left them out.) Now suppose you have another object from the same class:

After the statement "`b = a;`", you want this:

This is obvious different from just executing the statement "`b.x = a.x;`" which will give this picture instead:

where `b.x` will point to what `a.x` is pointing to. As mentioned in class, when `a` calls the destructor, the memory `a.x` is pointing will be released back to the heap and `b.x`

will be pointing to memory in the heap that has not been allocated:

so that when `b` called the destructor an error will occur when we try to do a `delete []` on `x` causing a double free error. (We already talked about this in class.)

The correct sequence of events for "`b = a;`" is depicted below. We start with this:

Step 1. First release the memory `b.x` is pointing to:

(We executed "`delete [] b.x`". `b.x` still have the same address as before but that array's memory is not allocated to any pointer and belongs to the heap.)

Step 2. Allocate enough memory for `b.x` to point to

(I'm writing `?` to denote garage integer values. Of course the array `b.x` points to is an integer array and do hold integer values.)

Step 3. Copy values `a.x` is pointing to array `b.x` is pointing to:

TADA! That's it. But there's a problem. This is a very subtle problem. Suppose the above is the sequence of events in `operator=` when we execute "`a = b;`". If `a` and `b` are distinct objects, then the above operations are in fact correct. But what if `a` and `b` are actually the same object?? This can happen when for instance we have this:

```
IntDynArr a(5);
IntDynArr & b = a; // b is a reference to a
```

Now let's track the above sequence of events for this scenario using our 3-step algorithm above. We start off with this:

Step 1. First release the memory `b.x` is pointing to (same as releasing `a.x` since `b` is a reference to `a`):

Step 2. Allocate enough memory for `b.x` to point to (same as allocating memory for `a.x`)

Step 3. Copy values `a.x` is pointing to into the array `b.x` is pointing to (same as copying `a.x` to `a.x`!!!)

Of course this means that you have lost all the original values of `a`. Now what?? Well, to correct the above, if `a` and `b` are the same objects (for instance if `b` is a reference to `a` or `a` is a reference to `b`), then we don't even want to perform anything at all since ... well ... they are the same already. This means that before you perform the above "`delete []`", "`new []`", copy values operations, you need to check if the objects are

the same. And you can check if the objects are the same by looking at their memory addresses. Here's then the corrected `operator=` (the same idea applies to the copy constructor):

```
IntDynArr & IntDynArr::operator=(const IntDynArr & b)
{
    if (this != &b)
    {
        ... // do something only if *this and b are different
    }
    return (*this);
}
```

This is the general template for `operator=` for all classes. (In most cases, the return type should be `const IntDynArr &`, but it wouldn't be wrong to return `IntDynArr &`. It depends on what you want to do after `operator=`.)

This is a very subtle error so make sure you remember it.

Since the copy constructor and `operator=` are very similar, parts of their code will be the same. Do not duplicate code. You can create a function that both the copy constructor and `operotor=` can both use.

Size and Capacity

The size of the array the `x` instance variable is pointing to is `capacity`. The instance variable `size` actually indicates the number of elements in the array that are valid. When you call the constructor

```
IntDynArr a(5);
a.resize(5);
for (int i = 0; i < 5; i++) a[i] = i;
```

`a.x` points to an array of 5 integers. Both `size` and `capacity` are set to 5.

Now when we call

```
a.resize(3);
```

we get the following picture:

Although `a.x` still points to an array of 5 integers, only the first three make up the array `a` is modeling. `a.x[3]` and `a.x[4]` are are not considered part of the array. Accessing them should be considered an incorrect operation. We are not enforcing the check now but later I will show you how to generate an error (called an "exception" object which can cause a program to halt).

Now suppose we call

```
a.resize(4);
```

we get this picture:

and when we call

```
a.resize(5);
```

we get the original picture:

But what if we request for more memory such as:

```
a.resize(6);
```

the end result of this call can be described by this picture:

(where ? denotes garbage value).

Note that the `capacity` is 12, i.e., twice of `size`. First of all let me tell you how to achieve this. Next I will explain why you try to grab so much memory, i.e., 12 and not just 6.

The above `resize` operation can be achieved by doing this. We start off with this:

Step 1. Declare a local `int` pointer, say `p`, to point to an array with 2*6 integers and set `capacity` to 2*6:

Step 2. Copy old values from the array `a.x` points to into the array `p` points to:

Step 3. Deallocate memory `a.x` is pointing to (in order to avoid memory leak):

Step 4. Point `a.x` to memory `p` is pointing to (i.e., `x` and `p` have the same address):

Step 5. Set the size to 6:

When you return from the resize method `p` is destroyed (since it's a local variable declared in the scope of `resize`):

Note that when you `resize`, values in the array are preserved whenever possible.

The above refers to `resize` in the context of a new larger size which is larger than the `capacity`. In this case, you always allocate memory for *twice* the new size.

Now why do you want to do something like that? Why not just allocate memory for the exact new size? Suppose you are in a situation where you `resize` by 1 more each time. For instance you want to execute `a.resize(6), a.resize(7), a.resize(8),` etc. Then the memory deallocation and allocation will be too costly. The above method of grabbing more memory than needed is to save on CPU time for memory allocation and deallocation. After the resize operation, if you need more memory for the array, you must have twice as much as you need.

What happens when you `resize` to a *smaller* size? If the new `size` is smaller than 1/3 of the `capacity`, you have to deallocate so that the new `capacity` is again twice the actual size needed. Let's do an example.

```
IntDynArr a(12);
a.resize(12);
for (int i = 0; i < 12; i++) a[i] = i;
```

The picture at this point is

Now suppose we execute:

```
a.resize(3);
```

Note that 3 is less than 1/3 of 12 (which is 4). The `resize` method should do this:

The array that `a.x` points to is now smaller and the capacity is again twice the new size.

If instead of the above we execute this:

```
a.resize(5);
```

we get the following picture instead:

In summary, if you `resize` and the new size is larger than the available capacity, you need to deallocate–and–allocate where the new array has a capacity that is twice the new size. If the `resize` is requested for a new size that is less than 1/3 of original capacity, you also have to deallocate–and–allocate so that the new array (again) has a capacity that is twice the new size. Deallocate-and-allocate is sometimes called reallocation.

[NOTE: Some dynamic array classes reallocate with a new capacity that is 1.5 times of the new size instead of 2 times.]

The methods in your class must follow the following rules on size and capacity:

- The amount of memory (i.e. the capacity) allocated by constructors is the capacity passed in and the size is set to 0.
- Except for `operator[]`, all other relevant methods and operators must change the capacity of the object according to the above rule. In other words, there is no change in the memory allocation when the new size is between capacity/3 and capacity; but when the size is $<$ capacity/3 or $>$ capacity, memory is re-allocated so that the new capacity is twice the new size. This applies to `resize()`, `operator+=`, `operator=`, `insert()`, and `remove()`.

A Common Error

A very common error I see is that students simply change the value of `capacity` assuming that the the point x will point to an array of different size. If your object has point x pointing to an array of 100 integers (and capacity is set to 100), obviously simply by changing `capacity` 1000 does not mean that x has valid access to 1000 integers in the heap!!!

That's like saying if you have a bathtub labeled 50 gallons and you simply took black paint and change the label to 500 gallons, then magically your bathtub will hold 500 gallons!!! That obviously is not true!!! You have to change the bathtub to actually achieve a new capacity!!!

Likewise if you want your `x` (in the object) to point to an array of with 1000 integers instead of 100, you have to request for 1000 from the heap and point your `x` to that new array in the heap (and of course making sure that the memory for the original 100 was deallocated properly.)

"Default" Value

Take note of the following cases:

- There is no check on array bound when `operator[]()` is called. (In other words in such cases, the program can cause an error.) You can however choose your own default value for testing purposes.
- If an array is resized, your class need not assign values to the extra elements, i.e., their values can be random. You can however choose your own default value for testing purposes.

Problems with Heap Usage

There are several tools for debugging programs with incorrect memory (or heap) usage. For instance you might want to google "c++ debug memory leak" or more specifically "c++ g++ debug memory leak" or "c++ .net studio debug memory leak". (I've already talked about asan, the address sanitizer tool from google. I've also talked about gdb.)

But the most important thing for you is to analyze carefully what your code should do and write your code carefully rather than dive in without careful thought and then hope that memory debugging tools will save you. You will find that memory debugging a badly written piece of code can take a lot more time than just simply writing better and correct code to begin with.

If you have to debug (and you will) your best friend when it comes to debugging and tracing code is the print statement. (For more complex systems, debugging tools are frequently more useful than printing.)

To trace where a statement crashes your program, all you need to do is to insert print statements.

```
... some code ...

std::cout << "here ... 0" << std::endl;

... some code ...

std::cout << "here ... 1" << std::endl;

... some code ...
```

If you don't see the second print statement when your program crashes, then of course the bug is between the two print statements. You then insert another print statement in the middle:

```
... some code ...

std::cout << "here ... 0" << std::endl;

... some code ...

std::cout << "here ... 0.5" << std::endl;

... some code ...
```

```
std::cout << "here ... 1" << std::endl;

... some code ...
```

This will allow you to "squeeze" the bug to find out it's exact location very quickly using the above "binary search" method.

One very important thing to note is that your program might crash before the print content printed arrived on your console window or terminal shell. To ensure that the printed content arrives before your program shuts down, you must print `std::endl`:

```
std::cout << "here ... 0" << std::endl; // GOOD FOR DEBUGGING!!!
```

and not this:

```
std::cout << "here ... 0" << '\n'; // BAD FOR DEBUGGING!!!
```

Test File

The following is only a skeleton test code. Complete it and test your library thorougly.

a10q01/skel/testIntDynArr.cpp

```
/*****************************************************************************
 * File  : testIntDynArr.cpp
 * Author:
 * Date  :
 *
 * Description: This is the test program for the IntDynArr class.
 *****************************************************************************/

#include <iostream>
#include "IntDynArr.h"

#define SIZE(x) (sizeof(x)/sizeof(int))


void test_IntDynArr()
{
    IntDynArr a;
    std::cin >> a;
    std::cout << a << std::endl;
}


void test_IntDynArr_size()
{
    int size;
    std::cin >> size;
    IntDynArr a(size);
    std::cout << a << std::endl;
}


void test_IntDynArr_array()
{
    int x[] = {1, 2, -4, 0};
    IntDynArr a(SIZE(x), x);
    std::cout << a << std::endl;
}


void test_print()
{
    int x[] = {1, 2, -4, 0};
    IntDynArr a(SIZE(x), x);
    std::cout << a << std::endl;
```

```
}


void test_size()
{
    IntDynArr a;
    std::cin >> a;
    std::cout << a.get_size() << std::endl;
}


void test_get_capacity()
{
    IntDynArr a;
    std::cin >> a;
    std::cout << a.get_capacity() << std::endl;
}

void test_eq()
{
    IntDynArr a, b;
    std::cin >> a >> b;
    std::cout << (a == b) << std::endl;
}

void test_plus_eq()
{
    IntDynArr a, b;
    std::cin >> a >> b;
    std::cout << (a += b) << ' ';
    std::cout << a << ' ';
    std::cout << ((a += b) += b) << std::endl;
}

void test_remove()
{
    IntDynArr a;
    int i;
    std::cin >> a >> i;
    std::cout << a << ' ' << a.remove(i) << std::endl;
}

int main()
{
    int option;
    std::cin >> option;
    switch (option)
    {
        case 1:
            test_IntDynArr();
            break;
```

```
        case 2:
            test_IntDynArr_size();
            break;
        case 3:
            test_IntDynArr_array();
            break;
        case 7:
            test_eq();
            break;
        case 17:
            test_remove();
            break;
        case 19:
            test_print();
            break;
    }

    return 0;
}
```

The test option numbering follows the same numbering as in the explanation earlier.