CS 2401 Spring 2023

Lab 4: Dynamic Arrays and the Big 3

Due: Friday 2/17 at 11:59 PM

(You can type your answers on the answer sheet so that it can be submitted electronically.)

This lab explores two important aspects of dynamic memory management. The first part deals with "The Big 3," the destructor, copy constructor and assignment operator that need to be overloaded for any class that holds dynamic memory. The second part deals with the *bad_alloc* exception and how it can be handled.

Begin by making a separate directory for this assignment. In that directory you will be using the main that I have given you and finishing the Numbers class, both the class declaration and the function implementations. (For this lab function implementations can be done under the class declaration, in the same file.)

The class, *Numbers*, that you declare will have these private variables:

```
unsigned long * data;

std::size_t used;

std::size_t capactity;
```

The functions should be:

1. A default constructor that sets up an initial array of 5 unsigned longs, sets capacity to 5 and used to 0.
2. An `add` function that receives an unsigned long as a parameter and puts it into the next available spot.
3. A `resize` function that is only called when the add function checks for and discovers that used == capacity. It should increase the size of the array by five.
4. A `remove_last` function that takes out the last thing added to the array. (All you need to do for this is to decrement the *used* counter. You do **not** need to change the capacity.)
5. A `display` function that prints out all the numbers in the array, separated by spaces.

Declare these functions in the class declaration and implement them immediately below that. (You do not have to use two files.)

The main does the following:

- `#include "numbers.h"` at the top.
- Declares two objects of your *Numbers* class, `N1` and `N2`.
- Adds the numbers 2 4 6 8 10 12 14 into `N1`. (If this crashes there's a problem with your resize().)
- Calls `N1.display();` to confirm that your numbers are in there.
- Assigns N2 to be a copy of N1: `N2 = N1;`
- Calls `N2.remove_last();` four times.
- Adds the numbers 5 10 15 into N2.
- Calls `N1.display(); // notice that this is the first object`
  `// not the one you just put numbers in`

On your **Answer sheet** write the answers to the following:

1. What do you see?
2. Is this a problem and why?
3. What caused this to happen?

Now, reopen the file where you defined your class, and add to your class an overloaded assignment operator. Remember that an assignment operator has three parts:

1) check for self-assignment

2) delete the existing array

3) create a new array the same size as the one you're copying from, copy the values for used and capacity, and then copy all the data from the other array into this new one. (The new array will be held by the pointer of the object of which this operator is a member.)

Compile and run the program again without changing anything in the main.

On your **Answer sheet** write the answers to the following:

4. What do you see?
5. Is it different from what you saw before?
6. What caused this to happen – why is it different?

Part Two:

Now, in the main, uncomment this for part two.

```
unsigned item = 0;
try{
     while (item < 5){
           Numbers N3; // five containers are going to be created
           for (int i = 0; i < 100; i++){
                N3.add(item);
           }
           cout << N3.reveal_address();
           ++item;
      }
catch (bad_alloc){
          cout << "Memory failure after adding " << item << endl;
}
```

Note that `reveal_address` is NOT something that we would normally do - the application programmer has no reason to need to know the address of the dynamic array, but I have included it here to help you see what is going on.

On your **Answer sheet:**

7. Write down the five addresses that are output and the byte count output.
8. Can you tell how many bytes they are apart? Write down your best estimate. (Remember that these addresses appear in hexadecimal.)

Eventually this array would eat up all the computer's memory and a bad_alloc exception would be "thrown" (and caught). Because modern computers have so much memory, we might wait a very long time for this to happen.

The solution is to write a destructor for our class. Add a destructor now and in it you should include, in addition to the correct `delete` command and some message you want to print:

```
byte_count = byte_count – (capacity * sizeof(unsigned long));
```

Compile the program again. Run the program again.

On your **Answer sheet:**

9. What addresses did you see this time and how much are they apart from each other?
10. What is the byte_count?
11. Explain why adding the destructor resulted in this different behavior.

Please submit your source code and your answer sheet.