DTS Lab 1  
dynamic array

Contents

[Objective 2](#_Toc444512592)

[Data Members 2](#_Toc444512593)

[Prototypes 2](#_Toc444512594)

[Desired Output 4](#_Toc444512595)

[Challenge 6](#_Toc444512596)

[Questions 8](#_Toc444512597)

[Submission 9](#_Toc444512598)

# Objective

Implement a templated class,*DynArray,* that will encapsulate the dynamic array functionality discussed in the Day 1 lecture materials.The default expansion rate should be double the current capacity with a minimum of 1.

Place all your code in a file named *DynArray.h*

# Data Members

The DynArray class will have the following private data members:

Type \* array = nullptr;  
unsigned int Size = 0, Capacity = 0;

# Prototypes

The DynArray class will have the following public interface:

/////////////////////////////////////////////////////////////////////////////  
// Function : Default Constructor  
// Notes : Doesn't do anything since data members were initialized above.  
// Still needs to exist due to copy constructor.  
/////////////////////////////////////////////////////////////////////////////  
DynArray();  
  
/////////////////////////////////////////////////////////////////////////////  
// Function : Destructor  
// Notes : cleans up any dynamic memory (can call clear())  
/////////////////////////////////////////////////////////////////////////////  
~DynArray();  
  
/////////////////////////////////////////////////////////////////////////////  
// Function : Copy Constructor  
// Notes : Call the assignment operator  
/////////////////////////////////////////////////////////////////////////////  
DynArray(const DynArray<Type>& that);  
  
/////////////////////////////////////////////////////////////////////////////  
// Function : Assignment Operator  
// Notes : Perform deep copy of 'that' internal array  
/////////////////////////////////////////////////////////////////////////////  
DynArray<Type>& operator=(const DynArray<Type>& that);  
  
/////////////////////////////////////////////////////////////////////////////  
// Function : operator[]  
// Parameters : index - the index to access  
// Return : Type & - returns the array element at [index]  
// Notes : performs no error checking  
/////////////////////////////////////////////////////////////////////////////  
Type& operator[](const unsigned int index);

/////////////////////////////////////////////////////////////////////////////  
// Function : operator[] const version  
// Notes : Same as above but works for const DynArray objects  
/////////////////////////////////////////////////////////////////////////////  
const Type& operator[](const unsigned int index) const

/////////////////////////////////////////////////////////////////////////////  
// Function : size  
// Returns : int - returns the Size data member  
/////////////////////////////////////////////////////////////////////////////  
unsigned int size() const;

/////////////////////////////////////////////////////////////////////////////  
// Function : capacity  
// Returns : int - returns the Capacity data member  
/////////////////////////////////////////////////////////////////////////////  
unsigned int capacity() const;  
  
/////////////////////////////////////////////////////////////////////////////  
// Function : clear  
// Notes : deletes any dynamic memory and resets size and capacity to 0  
// and array to nullptr  
/////////////////////////////////////////////////////////////////////////////  
void clear();  
  
/////////////////////////////////////////////////////////////////////////////  
// Function : append  
// Parameters : item - the item to be appended to the next open spot  
// Notes : this function will append the item to the next open spot. if   
// the array is full first, call reserve() to double the array, then  
// append the item.  
/////////////////////////////////////////////////////////////////////////////  
void append(const Type& item);  
  
/////////////////////////////////////////////////////////////////////////////  
// Function : reserve  
// Parameters : newCap - the new capacity  
// Notes : A) Default parameter of 0 should double the array capacity. In  
// this double case, check for existing capacity of 0 or non-0.  
// B) Parameter of lesser (non-0) or equal value to current capacity  
// should do NOTHING.  
// C) Parameter of higher value should be the newvalue for current  
// capacity.  
// Reallocates the array based on the new capacity amount (and copies over  
// the existing data)  
/////////////////////////////////////////////////////////////////////////////  
void reserve(const unsigned int & newCap = 0);

# Desired Output

Compile and run your code with the DTSLab1.cpp file provided via FSO. Your console output should match the following block identically:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\*\* LAB 1: \*\*  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
\*\*\* TEST 1 \*\*\*  
{ } Size 0 Capacity 0  
{ 0 } Size 1 Capacity 1  
{ 0 4 } Size 2 Capacity 2  
{ 0 4 8 } Size 3 Capacity 4  
{ 0 4 8 12 } Size 4 Capacity 4  
{ 0 4 8 12 16 } Size 5 Capacity 8  
{ 0 4 8 12 16 20 } Size 6 Capacity 8  
{ 0 4 8 12 16 20 24 } Size 7 Capacity 8  
  
\*\*\* TEST 2 \*\*\*  
{ 0 4 8 12 16 20 24 } Size 7 Capacity 16  
{ 0 4 8 12 16 20 24 } Size 7 Capacity 16  
{ } Size 0 Capacity 4  
{ 0 } Size 1 Capacity 4  
{ 0 4 } Size 2 Capacity 4  
{ 0 4 8 } Size 3 Capacity 4  
{ 0 4 8 12 } Size 4 Capacity 4  
{ 0 4 8 12 16 } Size 5 Capacity 8  
{ 0 4 8 12 16 20 } Size 6 Capacity 8  
{ 0 4 8 12 16 20 24 } Size 7 Capacity 8  
  
\*\*\* TEST 3 \*\*\*  
{ } Size 0 Capacity 10  
  
\*\* TEST 4 \*\*\*|  
{ 1 2 3 5 } Size 4 Capacity 4  
  
\*\*\* TEST 5 \*\*\*  
{ 0 4 8 12 16 20 24 } Size 7 Capacity 8  
{ } Size 0 Capacity 0  
{ 0 4 8 12 16 20 24 } Size 7 Capacity 8  
{ } Size 0 Capacity 0  
  
\*\*\* TEST 6 \*\*\*  
{ } Size 0 Capacity 0  
{ } Size 0 Capacity 1  
{ } Size 0 Capacity 2  
{ } Size 0 Capacity 4  
{ 0 1 } Size 2 Capacity 4

\*\*\* TEST 7 \*\*\*  
{ 0 1 2 3 } Size 4 Capacity 4  
  
\*\*\* TEST 8 \*\*\*  
{ 0 4 8 12 16 20 24 } Size 7 Capacity 8  
{ 28 } Size 1 Capacity 1  
{ 28 } Size 1 Capacity 1  
  
\*\*\* TEST 9 \*\*\*  
{ 1.1 2.2 3.3 4.4 5.5 } Size 5 Capacity 8  
{ 1.1 2.2 3.3 4.4 5.5 } Size 5 Capacity 8  
  
\*\*\* TEST 10 \*\*\*  
{ 0 4 8 12 16 20 24 } Size 7 Capacity 8

# Challenge

Add the following functions to your DynArray class. These will allow the user to insert or remove single or multiple items from valid indices in the array.   
\*these functions should only insert or remove at valid indices.

/////////////////////////////////////////////////////////////////////////////  
// Function : insert  
// Parameters : val - the value to insert  
// index - the index to insert at  
// Notes : if the array is full, this function should expand the array at   
// the default expansion rate (double the capacity, 1 minimum)  
/////////////////////////////////////////////////////////////////////////////  
void insert(const Type val, const unsigned int index)

/////////////////////////////////////////////////////////////////////////////  
// Function : insert  
// Parameters : val - the items to insert  
// n - the number of items to insert  
// index - the index to insert at  
// Notes : if the array is full, this function should expand the array at   
// the default expansion rate (double the capacity, 1 minimum)   
// before inserting  
/////////////////////////////////////////////////////////////////////////////  
void insert(const Type \* val, const unsigned int n, const unsigned int index)

/////////////////////////////////////////////////////////////////////////////  
// Function : remove  
// Parameters : index - the index to remove from  
// Notes : this function removes one item from the specified index  
/////////////////////////////////////////////////////////////////////////////  
void remove(const unsigned int index)  
  
/////////////////////////////////////////////////////////////////////////////  
// Function : remove  
// Parameters : index - the index to remove from  
// n - the number of items to remove  
// Notes : this function removes multiple items from the specified index  
/////////////////////////////////////////////////////////////////////////////  
void remove(const unsigned int index, const unsigned int n)

To test these functions, we'll use [the same .cpp file as before](file:///\\studentvfiler\DTS\out\Day%201), but you will need to change the CHALLENGE\_ACCEPTED macro from 0 to 1.

Compile and run your code now and the first 9 tests should remain unchanged, but output for tests 10 and 11 will be added. if your code works correctly, the new output should match the output below

\*\*\* TEST 11 \*\*\*  
{ } Size 0 Capacity 0  
{ 99 } Size 1 Capacity 1  
{ 28 99 } Size 2 Capacity 2  
{ 24 28 99 } Size 3 Capacity 4  
{ 20 24 28 99 } Size 4 Capacity 4  
{ 16 20 24 28 99 } Size 5 Capacity 8  
{ 12 16 20 24 28 99 } Size 6 Capacity 8  
{ 8 12 16 20 24 28 99 } Size 7 Capacity 8  
{ 4 8 12 16 20 24 28 99 } Size 8 Capacity 8  
{ 0 4 8 12 16 20 24 28 99 } Size 9 Capacity 16  
{ 0 1 2 3 4 8 12 16 20 24 28 99 } Size 12 Capacity 16  
  
\*\*\* TEST 12 \*\*\*  
{ 0 1 2 12 16 20 24 28 99 } Size 9 Capacity 16  
{ 1 2 12 16 20 24 28 99 } Size 8 Capacity 16  
{ 2 12 16 20 24 28 99 } Size 7 Capacity 16  
{ 12 16 20 24 28 99 } Size 6 Capacity 16  
{ 16 20 24 28 99 } Size 5 Capacity 16  
{ 20 24 28 99 } Size 4 Capacity 16  
{ 24 28 99 } Size 3 Capacity 16  
{ 28 99 } Size 2 Capacity 16  
{ 99 } Size 1 Capacity 16  
{ } Size 0 Capacity 16

# Questions

What data members do you think this class will need?

can you implement the reserve and append functions in such a way that data is copied only once?

Where are the potential memory leaks?

Explain why the output for test 5 is

\*\*\* TEST 5 \*\*\*  
{ 1 2 3 5 } Size 4 Capacity 4

and not

\*\*\* TEST 5 \*\*\*  
{ 1 2 3 4 5 } Size 5 Capacity 8

# Submission

To submit the lab assignment:

1. Clean, build, and run DTSLab1.cpp with your DynArray.h file in Visual Studio (debug mode).
   1. clear up any warnings you encounter.
   2. verify that your output is correct by comparing it to the lab document's Desired Output section, line-by-line.
   3. ensure there are no memory leaks.
2. Submit the 'DynArray.h' file via FSO.