For this assignment, you will implement a class that provides an automatically-resizing array. The array will automatically grow or shrink, as needed, according to the following rules:

- if the array is full when an insertion is needed, the dimension of the array will be doubled
- after an ~~insertion~~ deletion, if the array is less than 1/3 full, the dimension will be reduced to half its current value

These rules are similar to those that are most likely employed by the `vector` type in Standard C++. It turns out that these rules provide for very efficient insertions and deletions, on average. The growth rule buys a sensible amount of extra space for future expansion, and the shrinking rule leaves reasonable slack for future expansion.

To make it possible to easily change the type of data that may be stored in the array, the header file for the class includes a `typedef` statement that will be modified to specify the desired type of data.

Your implementation must conform to the class declaration shown below:

```
// DynArray.h
#ifndef DYNARRAY_H
#define DYNARRAY_H
#include <new>

typedef int T;   // specify the data type that will be stored

class DynArray {
   friend class Poirot;

private:
   unsigned int Dim;                          // dimension of the array
   unsigned int Usage;                        // number of cells in use
   T*           List;                         // pointer to array

   void Resize(unsigned int newCapacity);     // reset the size of the array
   void shiftTowardTail(unsigned int Start);  // shift data one position up
   void shiftTowardFront(unsigned int Start); // shift data one position down

public:
   DynArray(unsigned int Dimension = 100);    // create DynArray
   unsigned int Capacity() const;             // report the dimension
   unsigned int Size() const;                 // report the usage
   bool Append(const T& Elem);                // insert data at tail
   bool Insert(const T& Elem, unsigned int Idx); // insert data at Idx
   bool Delete(unsigned int Idx);             // delete data at Idx
   T*   Get(unsigned int Idx);                // return pointer to data at Idx
   void Clear();                              // reset array to empty state
   DynArray(const DynArray& Source);          // copy constructor
   DynArray& operator=(const DynArray& RHS);  // assignment operator
   ~DynArray();                               // deallocate array
};

#endif
```

This time the entire implementation is your responsibility. As with the earlier deep copy homework, you will submit your implementation to the Curator where it will be compiled with a copy of the header file shown above and a test harness that will test your code.

The effect of each member function is described in a brief comment above. For some, a more detailed description is given later in this specification. Be sure to read that carefully. In some cases, there are several reasonable interpretations of the comments above, but you must follow the detailed descriptions.

You will submit a file in the following format:

```
// DynArray.cpp
#include "DynArray.h"
using namespace std;

// Implementations of the specified member functions:
```

Note that this file will contain ONLY your implementations of the class member functions that are specified.

Here are some additional details about the behavior of some of the member functions:

`bool Append(const T& Elem);`

> If the array is not full, `Append()` stores `Elem` in the first unused cell. `Append()` returns true or false according to whether the insertion is actually performed.

`bool Insert(const T& Elem, unsigned int Idx);`

> If the array is not full and `Idx` corresponds to a cell in the range `0` to `Usage`, `Insert()` stores `Elem` at the specified index, shifting elements as necessary to make room. `Insert()` returns true or false according to whether the insertion is actually performed.

`bool Delete(unsigned int Idx);`

> If `Idx` corresponds to a used cell, `Delete()` removes the data element there, shifting elements as necessary to close the resulting hole. `Delete()` returns true or false according to whether the deletion is actually performed.

`T*  Get(unsigned int Idx);`

> If called with an index that matches a used cell in the array, `Get()` returns a pointer to the data in that cell. If called with an invalid index, `Get()` returns `NULL`.

`void Clear();`

> `Clear()` resets the data members to the state they were in immediately after the `DynArray` object was created.

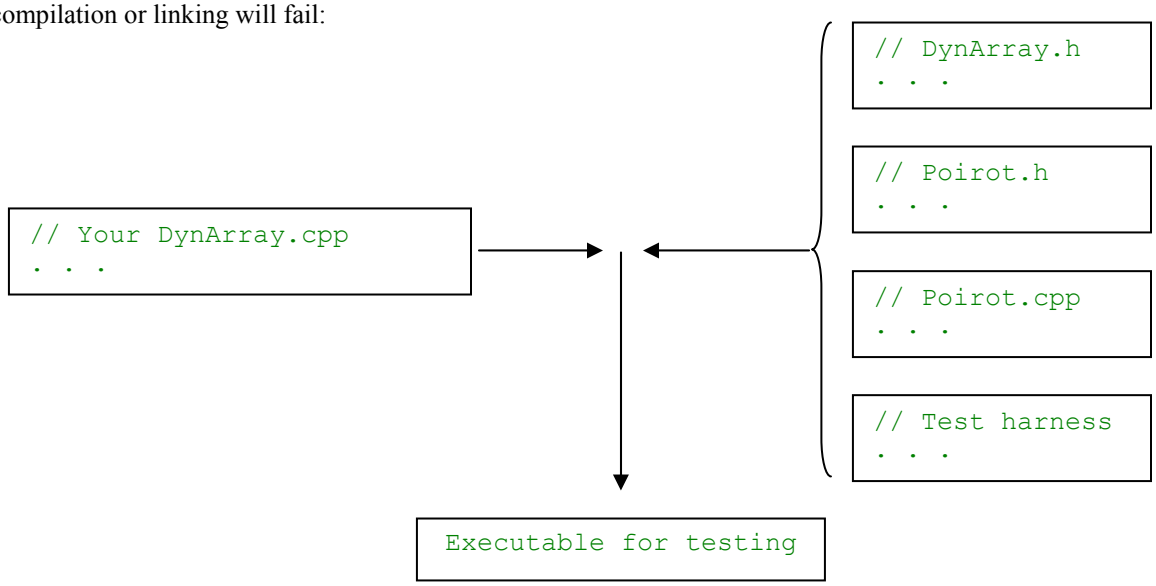
Poirot

You have probably noticed that the class header contains a forward declaration and `friend` declaration for a class named `Poirot`. This is a class that will be supplied on the Curator when your implementation is tested. Giving `Poirot` objects access privileges makes it possible for the test harness to use a `Poirot` object to directly examine the results of certain operations, possibly detecting logic errors and sidestepping runtime errors.

You do not have to implement `Poirot`, nor will any references to it occur in the code that you write. You are, of course, free to experiment with your own versions of `Poirot` if you like, but you must make sure that no references to that occur in the source file you submit to the Curator.

When you submit

What will happen when you submit? The mechanics are simple, but important since if you submit the wrong code then compilation or linking will fail:



The test driver will create instances of `DynArray` objects, initialize them in interesting ways, and thoroughly exercise the various ways of using them (which will depend on your implementation).

Evaluation

Do not waste submissions to the Curator to test your program! There is no point in submitting your program until you have verified that it operates correctly. If you waste all of your submissions because you have not tested your program adequately then you will receive a low score on this assignment. You will not be given extra submissions.

Your submitted program will be assigned a score, out of 100, based upon the runtime testing performed by the Curator System. There may or may not be a TA evaluation of your submission.

Submitting Your Program

You will submit this assignment to the Curator System (read the *Student Guide*), and it will be graded automatically. Instructions for submitting, and a description of how the grading is done, are contained in the *Student Guide*.

You will be allowed up to five submissions for this assignment. Use them wisely. Test your program thoroughly before submitting it. Make sure that your program produces correct results for every sample input file posted on the course website. If you do not get a perfect score, analyze the problem carefully and test your fix with the input file returned as part of the Curator e-mail message, before submitting again. The highest score you achieve will be counted.

The *Student Guide* and submission link can be found at: http://www.cs.vt.edu/curator/

Pledge

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the pledge statement (provided on the course website) in the header comment for your program.

Failure to include the pledge statement may result in a substantial grade penalty.