Assignment 6 – Dynamic Arrays

CS244

Due: Oct 30, 2013, 11:59 PM

Late penalties will be as described in the syllabus.

Overview
Create a C++ class DynArray and a program to test it.

General Objectives:

Explore:

C++ Class Design
Dynamic Arrays, Pointers, Memory allocation and de-allocation, plus some

First Step

In Linux in your …/Documents/Programs folder
Create a folder named ***A06***

This is necessary as you will compress the A06 folder and its contents for submission.

Second Step Check D2L

There will be starter code for this assignment. Likely posted near where this document was found. It may make things easier, and some of the files are explicitly required.

The files you will need for this assignment MUST be named:
DynArray.h DynArray.cpp DATester.cpp DAExcept.h makefile

with the corresponding classes properly defined and declared in each and your main() function located in DATester.cpp. DAExcept.h and makefile will be provided for you and ***should not*** require any changes. But do include them in your turn-in submission.

All files (excluding makefile and DAExcept.h) should have comments with the file name, your name, the last modified date, and a description of the file contents. Each function should also be appropriately commented in the header and implementation files. The body of each function may need comments for the less than obvious parts of the code (if any).

This assignment begins with background material and a worksheet (the worksheet will *not* be graded). The program code you turn in will be graded.

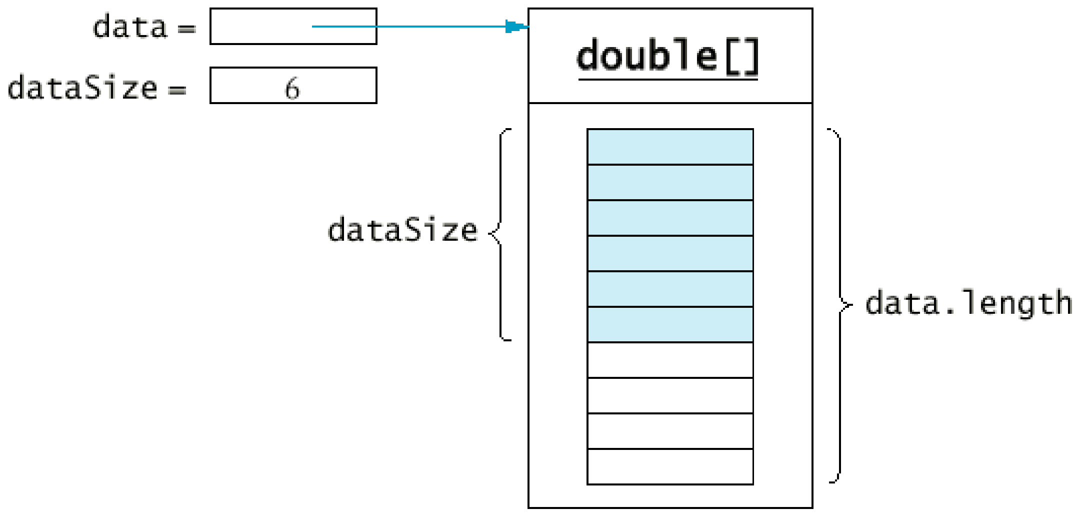
-=-=-

Introduction to the Dynamic Array (vector)

The array is the most primitive type of collection in the C++ programming language. An array is simply a block of values. The size of this block is determined when the array is created:

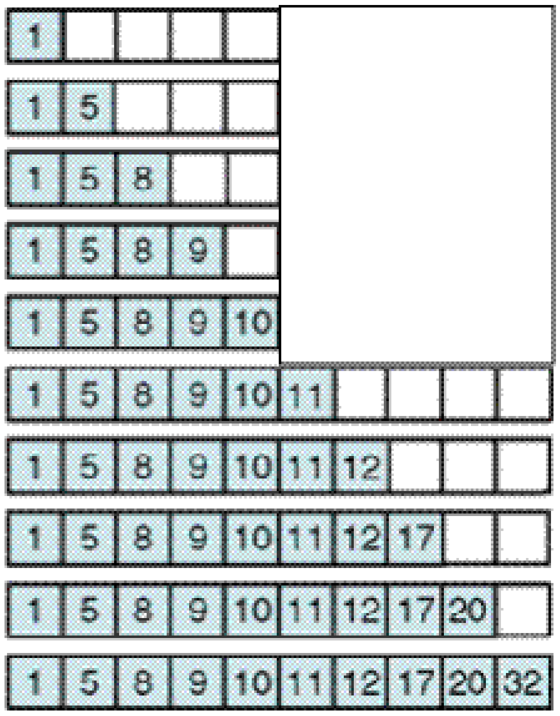
double points [30];   *// creates an array of 30 data points of type double*

A positive feature of the array is that it provides random access to values. Elements are accessed using the subscript [ ] operator, and the time it takes to access any element is no different from the time it takes to access another. However, a fundamental problem of the simple array is that the size must be specified at the time the array is created. Often the size cannot be easily predicted; for example if the array is being filled with values being read from a file. A solution to this problem is to use a *partially filled array*; an array that is purposely larger than necessary. A separate variable keeps track of the number of elements in the array that have been filled.



The *vector* (**Dynamic Array**) data type uses this approach. The array of values, current *size and capacity* of the collection is encapsulated within an object structure (think class header file). The size represents the number of elements in the array currently in use. The ***size* is different from the *capacity***. Capacity is the number of cells allocated for the array. Because the array is referenced by a pointer, an allocation routine must be called to set the initial size and create the initial memory area. A separate destroy (destructor) routine frees this memory.

The function ***pushBack*** places a new element at the end of the vector, increasing the size of the collection by one. The functions ***get*** and ***set*** replace the operations of subscript [ ] access and assignment in the array. A dynamic array (vector) can be used as a random access container, just like an array. The function ***size*** returns the number of elements (cells used) in the collection, which is not the same as the length (that is, capacity) of the internal array. The function ***capacity*** returns the currently allocated number of cells (capacity) of the dynamic array.



As new elements are added to the dynamic array (vector) it may happen that the *size* (the count of the number of elements) reaches the *capacity* (the number of elements in the array). At this point you need to create a new array twice the capacity of the current array and copy the values from the old array into the new array. The user then continues as before. Note that this increase in size occurs entirely within the Dynamic Array (vector) data abstraction. The user of the Dynamic Array (vector) does not know when this reallocation takes place. We have moved the action to a private separate function, named **growArray**. This method is not to be directly invoked by the user of the Dynamic Array (vector).

The task for this worksheet is to write the code for the Dynamic Array functions ***pushBack***, ***get***, ***set***, ***insert*** and **remove**. The member variable **m_size** represents the “logical” size; that is, the number of cells used in the vector.

First write the functions for ***get*** and ***set***.

```
// Function: get
// Precondition: the vector is not empty and index < size-1
// Postcondition: returns the value at index.
int DynArray::get(int index) const
{






}

// Function: set
// Precondition: 0 <= index < size-1
// Postcondition: update location index with value
void DynArray::set(int value, int index)
{








}
```

Next implement the functions ***pushBack*** and ***insert***. These will either add an element to the end of the vector or in the middle of the vector, increasing the size by one, and if necessary performing a reallocation to double the capacity of the array. You need to write the code to call ***growArray*** at the appropriate time in the constructors, ***pushBack***, ***insert***, and ***operator=*** methods.

Note: Some, if not all, of the growArray code has been provided for you in the starter code. Review it in DynArray.cpp to understand the logic

```
void DynArray::growArray(int n, bool copy)
{
      // Declare int pointer newArr to a new array of n elements


      // throw an exception if memory not allocated (done)
      if (newArr == NULL)
           throw memoryAllocationError("memory allocation failure");

      // if copy true, loop to copy elements in mp_array to newArr array




      // free memory currently used by mp_array



      // assign newArr to mp_array and update m_capacity to n



}
```

Also implement ***pushBack*** in DynArray.cpp

```
// Function: pushBack
// If pushBack pushes size past capacity double the size with growArray
// Postcondition: value appended at end of vector with size + 1
void DynArray::pushBack(int value)
{









}
```

Likewise implement the constructor, the copy constructor, the assignment operator, the friend comparison operator and the friend output stream operator<<

```
// Member function to overload the assignment operator
// Must make copy of dynamic array from rhs for *this ( left hand side of equals )
// Want *this = rhs to work, must do so element by element, return *this
DynArray& DynArray::operator= (const DynArray& rhs){ ... }

// compares two dynamic arrays for the same sizes and array values
bool operator== (const DynArray& lhs, const DynArray& rhs) { ... }

// output the elements in da.mp_array in order, return ostr
ostream& operator<< (ostream& ostr, const DynArray& da) { ... }
```

Questions (not graded but may show up in the future)

1. What is the algorithmic execution time for the operations **get** and **set** (Big-O)?

2. What is the algorithmic execution time for the operation **pushBack,** assuming there is sufficient capacity for the new elements.

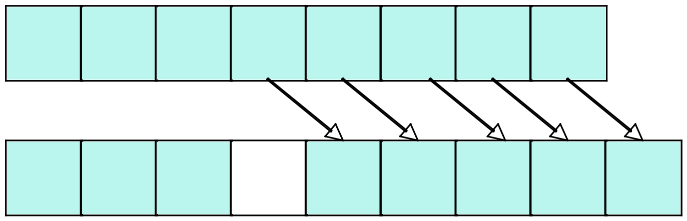
3. What is the algorithmic execution time for the internal method **growArray** when it doubles capacity?

4. Using as a basis your answer to 3, what is the algorithmic execution time for the operation **pushBack or insert** assuming that a new array must be created.

5. Explain the difference between the size and the capacity of a dynamic array (a vector).

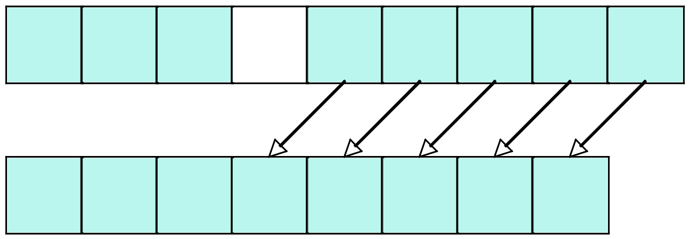
6. What happens inside a Dynamic Array when the size reaches the capacity? Does the user of the Dynamic Array know when this occurs?

In the first part of this assignment you started the implementation of the dynamic array (the vector) data type. To continue with the implementation we will add two new methods, a version of ***insert*** which inserts elements to the middle of the vector, and a method ***remove*** that deletes values and resizes a collection.



The ***insert*** operation will take an index value as the position to add the new element. To make space for the element all the values from that position to the end of the array must be moved up in order to make space for the new value. This will require a loop that will run from position index to the current data size. Think carefully about this loop. Should it run from index up to size, or the other direction?

The ***remove*** operation is just the opposite. Again, the element to be removed will be indicated by its index position. This time the values with index positions larger must be moved down.



Again this can be written as a loop.

Should this loop run from index upwards to **size**, or from **size** downward to index? Why?

Just as with the pushBack method, to insert a new value you must ensure that there is enough room for the new element, and call the function ***growArray*** that you have already written if there is not. Finally, make sure that the size of the collection is properly updated following the insertion. The code on the next page is the beginning of the implementation for these functions. What is the worst case algorithmic execution time for the ***insert*** operation? Can you identify any special cases where you can guarantee the execution time will be faster?

What is the worst case algorithmic execution time for the **remove** operation? Can you identify any special cases where you can guarantee the execution time will be faster?

```
// Function: insert
// insert value in n-th index position of the vector, mp_array.
// Postcondition: value at index with elements moved up and size+1
void DynArray::insert(int value, int index)
{











}

// Function: remove
// remove element at index location in the vector, mp_array
// Precondition: vector not empty else throws underflowError exception
// Postcondition: element at index removed and size--
void DynArray::remove(int index)
{











}
```

DynArray.h (Example Header File)

```
class DynArray
{
   public:
     DynArray(int capacity = 5);
       // constructor.
       // Postconditions: an empty vector with size = 0 and specified capacity.
     DynArray(const DynArray& obj);
       // copy constructor uses DynArray obj to create DynArray (*this)
       // Postcondition: creates current DynArray (*this) as a copy of obj
     ~DynArray();
       // destructor
       // Postcondition: the dynamic array vArr returns memory to the heap
     int get(int index) const;
       // Precondition: the vector is not empty and index < size-1
       // Postcondition: returns the value at index.
     void set(int value, int index);
        // Precondition: 0 <= index < size-1
        // Postcondition: update location index with value
     void pushBack(int value);
       //Postcondition: value appended at end of vector with size increased by 1
     void insert(int value, int index);
       // insert value in nth index position of the vector.
       // Postcondition: value at index n with elements moved up and size+1
     void remove(int index);
       // remove element at index location in the vector.
       // Precondition: vector is not empty, else throws underflowError exception
       // Postcondition: element at index removed and size--
     int size() const;
       // return current size of elements stored in Dynamic Array
     bool empty() const;
       // return true if vector is empty and false otherwise
     int capacity() const;
       // return the current capacity of the dynamic array
     void printVector() const;
       //print variable values and array for debugging

     DynArray& operator= (const DynArray& rhs);
       //Member function to overload the assignment operator
       //Must make copy of dynamic array from rhs for *this (left hand side of =)
     friend bool operator== (const DynArray& lhs, const DynArray& rhs);
       //compares two dynamic arrays for the same sizes and array values
     friend ostream& operator<< (ostream& ostr, const DynArray& da);
       // output the elements in vArr array in order.


    private:
     int m_capacity;        // amount of available space
     int m_size;            // number of elements in the list
     int* mp_array;         // the dynamic array of ints
     void growArray(int n, bool copy);
       // called by public functions if size of array needs to exceed vCapacity.
       // the vector capacity to n elements, copies the existing
       // elements to the new space if copy == true, and deletes
       // the old dynamic array. throws the memoryAllocationError
       // exception if memory allocation fails
};
```

The above is an example. It likely requires includes, pragmas, using, other statements added to work. Be careful about function and variable names. Use those provided in the starter code if there is a conflict.

All functions as described in the starter code header file (reflected above in the example header) must be implemented and working.
Be certain to de-allocate memory correctly (particularly in the destructor).

Hints for copy constructor and assignment operator:

Copy Constructor and operator= When working with an object that uses dynamic memory (memory from the heap), we must take care to create a new dynamic array with duplicate contents stored in its own location on the heap to copy one object into another. This happens two ways as shown in the code.

```
        DynArray v1(10);
        for(int i=0; i<20;i++)
              v1.push_back(i);
        DynArray v2(v1);        // invokes the copy constructor if exists
        DynArray v3(10);        // creates a third object
        v3 = v1;                // invokes operator= to copy v1 to v3
```

Without a copy constructor or overloading the operator= method, the above code would default to copying the data member values for m_size, m_capacity, and mp_array of v1 into the new DynArray object v2. The size and capacity values work fine, but the mp_array value copied is the memory location in the heap for the array of the original object v1. We need to create a new array on the heap for both v2 and v3 with the same values in the newly created DynArray object.

The copy constructor is called when a newly declared DynArray variable is set equal to an existing object. The existing object is assigned to rhs and v2 becomes *this.

```
// Copy Constructor
// Uses DynArray obj to create new DynArray(*this)
// Postcondition: makes current DynArray (*this) be a copy of obj
DynArray::DynArray(const DynArray& obj)
{
      // initialize mp_array to NULL and m_size to obj.m_size

      // use growArray to create a m_capacity sized array

      // copy items from obj.mp_array to mp_array
}

// operator=  (assignment operator)
// Member function to overload the assignment operator
// Make copy of dynamic array from rhs for *this (left hand side of =)
DynArray& DynArray::operator= (const DynArray& rhs)
{
      //use growArray set capacity vArr to rhs capacity with copy false

      // assign current object to have same size as rhs

      // copy items from rhs.vArr to vArr

      // return this object by dereferencing the this pointer
      return *this;
}
```

-=-=-=

Grading

100 points possible
At least 80 of these points will be determined by an “automatic grading” program.

The other 20 will be determined from code examination and/or further automated grading.
The majority of the points will be for correctly implementing the DynArray class.
Not as many points will be given for your main() testing function. That function and the majority of the testing cpp file is for you to estimate what your grade will be before turning it in. It is likely a separate testing file created by the instructor will be used to test your class(es). The better you write your own testing program the more likely you will pass the instructor’s.

Bonus Points

There may be in-class work that can be submitted for bonus points to be applied to this assignment. However, total score will not be increased beyond the 100 points possible.

Turn-In Directions

Correctly submitting your work is worth 0 points,
but if not done correctly will likely result in nothing to grade.

Preparation

In Ubuntu Linux browse to your A06 folder
Make sure your source code files are in the folder
Right click on the A06 folder,
Set the file name to be A06_*yourlastname*.tar.gz
where *yourlastname* is your last name
Example: if your last name is Gollygee
then the filename would be A06_Gollygee.tar.gz
Select compress
This should create the file named A06_ *yourlastname*.tar.gz

Submit

Submit the A06_ *yourlastname*.tar.gz file
to the correct course drop box in D2L