CS1321 Homework 5*

Jeffrey D. Oldham

2000 Mar 02

Due Thursday, 2000 Mar 16, at the beginning of class.

Contents

1 Revisions

2000Mar14: In lecture, we emphasized the importance of returning all allocated memory to the bit bucket when finished. After searching for two weeks for an adequate tool (and talking with a salesperson wanting to charge $8000 for a tool that did not find the most trivial memory leak!), we have found a primitive tool called `mtrace`.

Please consider testing your program for memory leaks. If `mtrace` indicates memory leaks, we will hand-inspect your program for memory leaks, deducting points if we find any.

To use the tool, following these steps:

1. Download the revised test-dynamicArray.cc and compile it. Let's assume the executable is named `a.out`.

2. Before running the executable, type

   ```
   declare -x MALLOC_TRACE=foo.txt
   ```

   in a shell. Instead of `foo.txt`, you can use any file name you desire.

3. Run the executable. Memory allocation and deallocation information is stored in the file called `foo.txt`.

4. To print memory leak information, type

   ```
   mtrace a.out $MALLOC_TRACE
   ```

   in the same shell.

For more information, see the info pages. For example, using emacs, type `Control-h i`, `m libc`, `m memory allocation`, and `m allocation debugging`.

Caveats:

1. Ignore any leak information not involving the words "new" or "delete." For example, ignore any leaks involving the word "exit."
2. Ignore the indicated line numbers. Just check your code for `new`s without corresponding `delete`s and vice versa.
3. Using STL `string`s may cause memory leak errors; we do not know why. Thus, for `show_op_counts`, we recommend using a C-string, not an STL string.

2000Mar13: Revised Oldham's conjecture for the naïve implementation from $n(n+1)/2$ to $n(n-1)/2$ copies. (Also, the doubling implementation requires a maximum of $2n$ element copies, not $3n$.)

2 Reading

Read chapter 4 of the textbook. Start reading chapter 5.

3 Problem Statement

You and possibly a CS1321 classmate are to implement a dynamic array class using arrays and dynamic memory using two different strategies. The classes will differ in their strategies for when to resize arrays. Then, please show that one implementation runs asymptotically faster than the other.

As always, please read through the entire assignment before beginning to code; timing your code might be easier if you read through that section first.

3.1 Introduction

A *dynamic array* is an array that grows and shrinks as more or less storage is needed. For example, the STL `vector` class supports dynamic array operations. As more elements are added to a `vector`, e.g., using `push_back`, it increases in size. If elements are removed, e.g., using `pop_back`, it decreases in size. Thus, users can avoid the requirement of specifying an ordinary array's maximum size at creation time.

In this homework, we will implement the dynamic array class named `dynamicArray` introduced during lecture.

3.2 Naïve Implementation Strategy

Our naïve implementation strategy for the `dynamicArray` class is that each object should store its n items in a dynamically-allocated array of size n. For example, an object holding 17 items will store them in a dynamically-allocated array of size 17. (Dynamic memory is sometimes called the *heap* or colloquially "the bit bucket.") To add one item to the end of the array, an array of size $n+1$ is allocated, the existing n items are copied to the new array, and the old array is returned to the bit bucket. The procedure to remove one item from the end of the array is similar. `dynamicArray` objects only support adding and removing elements from the "right" end of the array.

`dynamicArray` objects should support the operations listed in Table 1. If in doubt about a function's semantics, read about the corresponding `vector` function. `push_back` increases the number of items stored in the `dynamicArray` object, while `pop_back` decreases the number of items. Using `pop_back` on an array with no elements is undefined; you choose whether to check for this case or not. Similarly, `get` and `set` can optionally check for correct position values. Define a copy constructor, an assignment operator, and a destructor if necessary. (Hint: They are necessary. See also the textbook or lecture notes.)

Assume the dynamic array stores `int`s, but notice how easy it will be to change to a different type by changing the definition of `item_type`. Positions are numbered just as for ordinary arrays and `vector`s. The "leftmost" element is numbered 0 and the "rightmost" element has number $n-1$ if the array has n items.

As alluded to above, implement the class using `new`, `delete`, and arrays, not using `vector`s or other dynamic array classes defined by other people.

function prototype	example use	explanation
`dynamicArray(void)`	`dynamicArray v;`	create an object with no items
`length_pos`	`dynamicArray::length_pos i = v.size();`	type specifying array's length or a position within array
`item_type`	`dynamicArray::item_type i = v.pop();`	type specifying an array element
`void push_back(const item_type & item)`	`v.push_back(17);`	append the given item to the end of the array
`item_type pop_back(void)`	`int i = v.pop_back();`	remove the last element of the array and return it
`length_pos size(void)`	`dynamicArray::length_pos i = v.size();`	return the number of items in array
`item_type get(const length_pos i)`	`int i = v.get(0);`	return the item stored at the specified position
`void set(const length_pos i, const item_type & item)`	`v.set(0,3);`	store the second parameter in the position specified by the first parameter

Table 1: `dynamicArray` Member Functions and Types

3.3 Powers-of-Two Implementation

The naïve implementation allocates an array exactly the same size as the number of elements to hold. Thus, every time an element is added or removed, an entirely new array is allocated. If we permit an array to have a size different from its number of elements, we can do better using the "double or halve" heuristic, a common computer science rule of thumb:

When allocating a new array, either double or halve its size.

Initially, the array should have size one even though it has no elements. Whenever adding an additional element requires reallocating the array, double the array's size. Whenever an array becomes less than *one-quarter* full, halve its size. Thus, the array's size will always be a power of two.

For example, consider the following sequence of operations:

operation	nu. elements	array size	
`dynamicArray v;`	0	1	`new` an array
`v.push_back(3);`	1	1	
`v.push_back(4);`	2	2	`new` an array
`v.push_back(5);`	3	4	`new` an array
`v.push_back(1);`	4	4	
`int i = v.pop_back();`	3	4	
`int i = v.pop_back();`	2	4	
`int i = v.pop_back();`	1	4	
`int i = v.pop_back();`	0	2	`new` an array

Reimplement `dynamicArray` to incorporate this new strategy, but be sure to save your old code in a different file.

3.4 Timing Comparisons

Theoretician Jeffrey D. Oldham claims that the doubling implementation runs asymptotically faster than the naïve implementation. In fact, he makes an even more precise claim:

> Consider a sequence of n `push_back` operations. The naïve implementation requires n reallocations and copying of $n(n-1)/2$ elements. The doubling implementation requires $\log_2 n$ reallocations and a maximum of $3n$ element copies.

Experimentally prove his claim. Instrument both implementations to count the number of array reallocations and associated element copies. A *reallocation* is the process of allocating a new array, copying the existing array's elements to the new array, and destroying the old array. If you write your code in a modular form, this should occur in only one function. Also add a member function `show_op_counts` to both `dynamicArray` implementations that, given an `ostream` and a string representing the array's name, prints the array's statistics.

Run the timing program using both implementations. With your submission, provide three graphs with x-axes corresponding to the number of `push_back` operations and y-axes for the number of reallocations, number of element copies, and running times, respectively. Please plot these for both implementations and submit *on paper*; we are Microsoft-challenged. It might be interesting to try using a number of `push_back`s comparable to 16K, 32K, ... , 128K.

4 What Files Do I Need?

There are two provided files:

- `test-dynamicArray.cc` partially tests a `dynamicArray` class. You may want to write some more tests. This code assumes your `dynamicArray` class is in a file called `dynamicArray.h`.
- `time-dynamicArray.cc` exercises a `dynamicArray` implementation. This may be helpful when proving Jeffrey's claim. This code assumes your `dynamicArray` class is in a file called `dynamicArray.h`.

Submit your two implementations in files named `naive-dynamicArray.h` and `powers-dynamicArray.h`.

5 Rules

5.1 Programming Rules

As for previous homeworks, working with other people during your planning phase is encouraged. For this homework, you are permitted to write the code, i.e., program, with one other person in CS1321. To learn the material, both of you should be actively involved in the programming. Failure to do so will almost certainly hurt your comprehension of the material in the rest of the course.

5.2 Submission Rules

Each one- or two-person team of programmers should submit only your two completed implementations in files named `naive-dynamicArray.h` and `powers-dynamicArray.h`. Submit your experimental evidence *on paper*. You need not send any other files. Please send only text documents, do *not* send Microsoft Word documents, PDF documents, HTML documents, etc. **Please include both your names and email addresses at the top of your program.**

We will test your code using our own main() function. Please be sure it compiles without warning when using `g++ -Wall -pedantic`.

See the submission details for information how to email the programs. If a team of two are in different sections, submit exactly once to one of the two permissible email addresses.