CSCI 1321 (Principles of Algorithm Design II), Fall 2001

Homework 5[1]

Assigned: October 25, 2001.

Due: November 6, 2001, at 5pm.

Credit: 40 points.

> *Note:* The HTML version of this document may contain hyperlinks. In this version, hyperlinks are represented by showing both the link text, formatted like this, and the full URL as a footnote.

Contents

1 Problem statement **1**
- 1.1 Introduction . . . 2
- 1.2 What the class must provide . . . 2
- 1.3 Implementation . . . 3
- 1.4 Naive implementation strategy . . . 3
- 1.5 Powers-of-two implementation strategy . . . 3
- 1.6 Timing comparisons . . . 4

2 Checking for memory leaks **4**

3 What files do I need? **5**

4 What to turn in **6**
- 4.1 Source code . . . 6
- 4.2 Experimental evidence (graphs) . . . 6

1 Problem statement

You 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. You are then to 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.

[1] © 2001 Jeffrey D. Oldham (`oldham@cs.stanford.edu`) and Berna L. Massingill (`bmassing@cs.trinity.edu`). All rights reserved. This document may not be redistributed in any form without the express permission of at least one of the authors.

1.1 Introduction

A *dynamic array* is an array that grows and shrinks as more or less storage is needed. For example, the STL `vector` class[2] 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 a dynamic-array class, i.e., a primitive substitute for the STL `vector` class.

1.2 What the class must provide

`dynamicArray` objects should support the operations listed in the following table.

Function prototype	Example use	Explanation
`dynamicArray(void)`	`dynamicArray v;`	Creates 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_back();`	Type specifying an array element.
`void push_back(const item_type & item)`	`v.push_back(17);`	Appends the given item to the end of the array.
`item_type pop_back(void)`	`int i = v.pop_back();`	Removes the last element of the array and returns it.
`length_pos size(void)`	`dynamicArray::length_pos i = v.size();`	Returns the number of items in array.
`item_type get(const length_pos i)`	`int i = v.get(0);`	Returns the item stored at the specified position.
`void set(const length_pos i, const item_type & item)`	`v.set(0,3);`	Stores the second parameter in the position specified by the first parameter.

If in doubt about a function's semantics, read about the corresponding function in the STL `vector` class[3]. `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. (This might be a good place to use `assert()`.) Similarly, `get()` and `set()` can optionally check for acceptable position values (i.e., not bigger than the array's size). Define a copy constructor, an assignment operator, and a destructor if necessary. (Hint: They are necessary.)

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 STL `vector`s: The "leftmost" element is numbered 0 and the "rightmost" element has number $n-1$, where n is the number of items in the array.

[2] `http://www.sgi.com/tech/stl/Vector.html`
[3] `http://www.sgi.com/tech/stl/Vector.html`

1.3 Implementation

Your mission in this assignment is to define a class `dynamicArray` that provides the functions and types described in the preceding section. You are to implement the class using `new`, `delete`, and arrays, *not* using `vector`s or other library classes. You may, however, use the simple array class presented in lecture (`dArray.h`[4]) as a starting point.

The next sections discuss two possible implementation strategies. The first is somewhat less trouble to implement; the second is more efficient. You are to implement the class twice, once for each strategy, and compare their performance, as discussed later. Notice that you will need to be a little careful about maintaining two copies of your class definition, since the test and timing programs assume your class is defined in a file called `dynamicArray.h`. A reasonable approach is to keep each implementation in a separate subdirectory.

1.4 Naive implementation strategy

Our naive 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 (not from the left end or middle).

1.5 Powers-of-two implementation strategy

The naive 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	**Number of elements**	**Array size**	**Comments**
`dynamicArray v;`	0	1	makes a new array
`v.push_back(3);`	1	1	
`v.push_back(4);`	2	2	makes a new array
`v.push_back(5);`	3	4	makes a new 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	makes a new array

[4] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/SamplePrograms/dArray.h`

Note that the `size()` member function should continue to return the number of elements actually stored in the array, which may be different for this strategy from the array's physical size.

1.6 Timing comparisons

A theoretically-minded friend claims that the powers-of-2 implementation runs asymptotically faster than the naive implementation, and in fact makes an even more precise claim:

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

Experimentally prove this claim: Instrument your classes 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 (number of reallocations and element copies).

Run the timing program `time-dynamicArray.cpp`[5] using your implementation(s). This program allows you to measure running time and the numbers of reallocations and copies required for a specified number of `push_back()` operations, say n. Use it collect this information for several values of n (enough to see how running time, etc., changes as n increases); plot the results in the form of three separate graphs, one each for running time, number of reallocations, and number of copies. The running-time graph should plot running time (the y axis) versus number of `push_back()` operations (the x axis); the other two graphs should be similar. You may do this by hand or using any program that provides appropriate functionality. (I use `gnuplot`.) It might be interesting to try using a number of `push_back()`s comparable to $16000, 32000, \ldots, 128000$.

2 Checking for memory leaks

In lecture, we emphasized the importance of returning all allocated memory to the bit bucket when finished. There are commercial tools available for this purpose, but some are quite expensive. A free, if primitive, alternative is a tool called `mtrace`. You can use this tool by following these steps:

1. Compile the program to be checked for memory leaks (`test-dynamicArray.cpp`[6], for example). Suppose the executable is named `a.out`.

2. Before running the executable, type

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

   (You can replace `foo.txt` with any filename you like.)

3. Run the executable as you usually do. Memory allocation and deallocation information is stored in the file `foo.txt` (or whatever filename you chose in the step above).

4. To print memory leak information, type

[5] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/HW05/Problems/time-dynamicArray.cpp`
[6] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/HW05/Problems/test-dynamicArray.cpp`

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

(If your executable is not called `a.out`, replace `a.out` in the above command with the name of your executable.)

For more information, see the info pages. ("info pages", like "man pages", are a standard form of Unix/Linux documentation.) You can access the relevant pages by first typing `info` (to start a text-based program to browse info pages) and then typing `m libc`, `m memory allocation`, and `m allocation debugging`.

Some caveats:

1. Ignore any leak information not involving the words "new" or "delete." For example, ignore any leaks involving the words "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; the algorithm the STL `string` class uses to allocate memory does not clean up after itself properly. Thus, for `show_op_counts`, I recommend using a C-style string (array of characters) rather than an STL `string`.

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

3 What files do I need?

There are two provided files:

- `test-dynamicArray.cpp`[7] partially tests a `dynamicArray` class. You may want to write additional tests. This code assumes your `dynamicArray` class is in a file called `dynamicArray.h`.
- `time-dynamicArray.cpp`[8] exercises a `dynamicArray` implementation. This may be helpful in proving the claim about numbers of operations. This code assumes your `dynamicArray` class is in a file called `dynamicArray.h`.

You may also find the following sample programs useful:

- The class for dynamically-allocated arrays presented in lecture. File `dArray.h`[9] contains the class definition, and file `dArray-use.cpp`[10] shows simple examples of its use.
- The version of the "print in two columns" program that uses a dynamically-allocated array that is expanded as needed, `column-print-3.cpp`[11].

[7] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/HW05/Problems/test-dynamicArray.cpp`
[8] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/HW05/Problems/time-dynamicArray.cpp`
[9] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/SamplePrograms/dArray.h`
[10] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/SamplePrograms/dArray-use.cpp`
[11] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/SamplePrograms/column-print-3.cpp`

4 What to turn in

4.1 Source code

Submit your implementation(s) in files named `naive-dynamicArray.h` and `powers-dynamicArray.h`. You do not need to submit a main program; I will test your code using my own `main()` function. Submit this source code as described in the Guidelines for Programming Assignments[12]. For this assignment use a subject line of "cs1321 hw 5".

4.2 Experimental evidence (graphs)

It is probably easiest if you submit your graphs on paper, either in class or by dropping them off in my mailbox in the department office. You may also submit them electronically (i.e., via e-mail, as you do source code) if you choose a format I can read. Formats I know I can read are PostScript, PDF, GIF, and Microsoft Word (the last with some hassle). If you want to submit graphs in some other format, check with me first to make sure I will be able to read it.

[12] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/Notes/pgmguidelines/index.html`