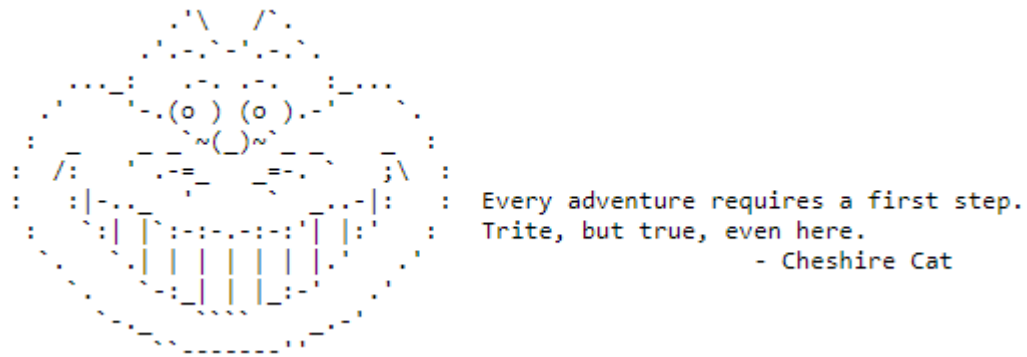
CS 15 Homework 1: ArrayLists



Introduction

In this assignment you will implement a version of the array list data structure discussed in class that contains characters.

Recall that an array list is a kind of list: An ordered collection of data values. "Ordered" here does not mean "sorted," it just means that, if there are items in the list, there is a distinct first element, second element, etc. We will use 0-based indexing, i.e., the first element in an array list will be element 0. Note that an array list cannot have "holes": if you remove the fifth element (element 4) from a 10-element array list, then there are 9 elements left, and their positions are 0 through 8.

You will write both the public and private sections of the `CharArrayList` class. The class definition will go in a file named `CharArrayList.h`; the class implementation will go in a file named `CharArrayList.cpp`. You will

also write test code in `unit_tests.h` (described below). Your goal is to implement a well-tested array list that a client could pick up and use in their own program.

We'll describe the (public) interface first, then give some some implementation specifics, and finally submission instructions.

Program Specification

Important Notes

- The names of your functions/methods as well as the order and types of parameters and return types must be exactly as specified. This is important because we will be compiling the class you wrote with our own client code!
- Any exception messages should likewise print exactly as specified and use the given error type.
- You may not have any other public functions.
- All data members must be private.
- You may **not** use any C++ strings in your `CharArrayList` implementation **except for:**
  - In the `toString()` and `toReverseString()` functions.
  - When throwing exception messages.
- You may **not** use `std::vector` or any other built-in facility that would render the assignment trivial.

Interface

Your class must have the following interface (all the following members are public):

- Define the following constructors for the `CharArrayList` class:
  - `CharArrayList()`

The default constructor takes no parameters and initializes an empty array list. This array list has an initial capacity of 0.

- `CharArrayList(char c)`
  The second constructor takes in a single character as a parameter and creates a one element array list consisting of that character. This array list should have an initial capacity of 1.
- `CharArrayList(char arr[], int size)`
  The third constructor takes an array of characters and the integer length of that array of characters as parameters. It will create an array list containing the characters in the array. This array list should have an initial capacity equal to the length of the array of characters that was passed.
- `CharArrayList(const CharArrayList &other)`
  A copy constructor for the class that makes a deep copy of a given instance.

Recall that all constructors have no return type.

- `~CharArrayList()`
  Define a destructor that destroys/deletes/recycles all heap-allocated data in the current array list. It has no parameters and returns nothing.

- `CharArrayList &operator=(const CharArrayList &other)`
  Define an assignment operator for the class that recycles the storage associated with the instance on the left of the assignment and makes a deep copy of the instance on the right hand side into the instance on the left hand side.

- `bool isEmpty() const`
  An `isEmpty` function that takes no parameters and returns a boolean value that is true if this specific instance of the class is empty (has no characters) and false otherwise.

- `void clear()`
  A `clear` function that takes no parameters and has a `void` return type. It makes the instance into an empty array list. For example if you call the `clear` function and then the `isEmpty` function the `isEmpty` function should return true.

- `int size() const`
  A `size` function that takes no parameters and returns an integer value

that is the number of characters in the array list. The size of an array list is 0 if and only if it isEmpty.

- `char first() const`
  A `first` function that takes no parameters and returns the first character in the array list. If the array list is empty it should throw an `std::runtime_error` exception with the message "cannot get first of empty ArrayList". Note that this exception message *does not* end with a newline.

- `char last() const`
  A `last` function that takes no parameters and returns the last element `(char)` in the array list. If the array list is empty it throws a `std::runtime_error` exception with the message "cannot get last of empty ArrayList". Note that this exception message *does not* end with a newline.

- `char elementAt(int index) const`
  An `elementAt` function that takes an integer index and returns the element `(char)` in the array list at that index. NOTE: Indices are 0-based. If the index is out of range it should throw a C++ `std::range_error` exception with the message "index (IDX) not in range [0..SIZE)" where IDX is the index that was given and SIZE is the size of the array list. For example: "index (6) not in range [0..3)" if the function were to be called using the index 6 in a size 3 array list. Note the braces and the spacing, and also note that there is no newline!

- `std::string toString() const` function
  A `toString` function that takes no parameters and has a `std::string` return type. It returns a string which contains the characters of the CharArrayList. The string will be formatted like this:

  ```
  [CharArrayList of size 5 <<Alice>>]
  ```

  where, in this example, 5 is the size of the array list and the elements are the characters `'A'`, `'l'`, `'i'`, `'c'`, `'e'`. The empty array list would be formatted like this:

  ```
  [CharArrayList of size 0 <<>>]
  ```

  Note: There is no newline after the last `]`.

Caution: The format of strings is essential to get exactly right, because your output will be verified automatically. There is no whitespace printed, except the single spaces shown between the words inside the square brackets (i.e. "CharArrayList 'space' of 'space'", etc.). The capitalization must be exactly as shown.

- `std::string toReverseString() const` function
  A `toReverseString` function that takes no parameters and has a `std::string` return type. It returns a string which contains the characters of the CharArrayList in reverse. The string will be formatted like this:

  ```
  [CharArrayList of size 5 <<ecilA>>]
  ```

  where, in this example, 5 is the size of the array list and the elements are the characters `'A'`, `'l'`, `'i'`, `'c'`, `'e'`. The empty array list would be formatted like this:

  ```
  [CharArrayList of size 0 <<>>]
  ```

  Note: There is no newline after the last `]`.

- `void pushAtBack(char c)`
  A `pushAtBack` function that takes an element `(char)` and has a `void` return type. It inserts the given new element after the end of the existing elements of the array list.

- `void pushAtFront(char c)`
  A `pushAtFront` function that takes an element `(char)` and has a `void` return type. It inserts the given new element in front of the existing elements of the array list.

- `void insertAt(char c, int index)`
  An `insertAt` function that takes an element `(char)` and an integer index as parameters and has a `void` return type. It inserts the new element at the specified index and shifts the existing elements as necessary. The new element is then in the index-th position. If the index is out of range it should throw a C++ `std::range_error` exception with the message "index (IDX) not in range [0..SIZE]" where IDX is the index that was given and SIZE is the size of the array list. NOTE: It is allowed to insert at the index after the last element. Note that the braces in this message are different from those in the `elementAt` range error.

- `void insertInOrder(char c)`
  An `insertInOrder` function that takes an element `(char)`, inserts it into the array list in ASCII order, and returns nothing. When this function is called, it may assume the array list is correctly sorted in ascending order, and it should insert the element at the first correct index. Example 1: Inserting 'C' into "ABDEF" should yield "ABCDEF" Example 2: Inserting 'A' into "ZED" should yield "AZED." You can rely on the built-in `<, >, <=, >=,` and `==` operators to compare two chars.

- `void popFromFront()`
  A `popFromFront` function that takes no parameters and has a `void` return type. It removes the first element from the array list. If the list is empty it should throw a `C++ std::runtime_error` exception with the message "cannot pop from empty ArrayList".

- `void popFromBack()`
  A `popFromBack` function that takes no parameters and has a `void` return type. It removes the last element from the array list. If the list is empty it should throw a `C++ std::runtime_error` exception initialized with the string "cannot pop from empty ArrayList".

- `void removeAt(int index)`
  A `removeAt` function that takes an integer index and has a `void` return type. It removes the element at the specified index. If the index is out of range it should throw a `std::range_error` exception with the message "index (IDX) not in range [0..SIZE)" where IDX is the index that was given and SIZE is the size of the array list.

- `void replaceAt(char c, int index)`
  A `replaceAt` function that takes an element `(char)` and an integer index as parameters and has a `void` return type. It replaces the element at the specified index with the new element. If the index is out of range it should throw a `std::range_error` exception with the message "index (IDX) not in range [0..SIZE)" where IDX is the index that was given and SIZE is the size of the array list.

- `void concatenate(CharArrayList *other)`
  A `concatenate` function that takes a pointer to a second `CharArrayList` and has a `void` return type. It adds a copy of the array list pointed to by the parameter value to the end of the array list the function was called from. For example if we concatenate `CharArrayListOne`, which

contains “cat” with `CharArrayListTwo`, which contains “CHESHIRE”, `CharArrayListOne` should contain “catCHESHIRE”. Note: An empty array list concatenated with a second array list is the same as copying the second array list. Concatenating an array list with an empty array list doesn't change the array list. Also an array list can be concatenated with itself, e.g concatenating `CharArrayListTwo` with itself, results in `CharArrayListTwo` containing “CHESHIRECHESHIRE”.

- `void shrink()`
  A shrink() function that takes no parameters and has a `void` return type. It reduces the object's memory usage to the bare minimum required to store its elements (i.e. it does not use any extra space).

You may add any private methods and data members. We particularly encourage the use of private member functions that help you produce a more modular solution.

Before you start writing any functions please sit down and read this assignment specification. Some of these functions do similar tasks. Perhaps it would be prudent to organize and plan your solution using the principles of modularity, e.g., helper functions. This initial planning will be extremely helpful down the road when it comes to testing and debugging; it also helps the course staff more easily understand your work (which can only help).

Also, the order in which we listed the public methods/functions of the `CharArrayList` class, is not the easiest order to implement them in. If you plan your functions out and identify the easy ones it will make your work easier and your final submission better.

If you are having issues planning out your assignment we encourage you to come in to office hours as early as possible.

JFFEs (Just For Fun Exercises)

If you complete the above functions, you may add the following functions. There is no extra credit, but they're fun and educational.

- `void sort()`
  A `sort` function that takes no input and has a `void` return type. It sorts the characters in the list into alphabetical order.

- `CharArrayList *slice(int left, int right)`
  A `slice` function that takes a left index and a right index and returns a pointer to a new, heap-allocated `CharArrayList`. The new array

list contains the characters starting at the left index and up to, but not including, the right index. If the first index is equal to or greater than the second, it returns a new, empty array list. The left index must be in the range [0..SIZE) and the right index must be in the range [0..SIZE] where SIZE is the size of the array list. Since the right index is not included in the final slice, requesting a slice where the right index is 1 index past the last element is still a valid request. If the either index passed is out of range the function should throw a `std::range_error` with an appropriate message.

Implementation Details

Copy the starter files from `/comp/15/files/hw1` to get the exception examples and program files with header comments:

- `CharArrayList.h`
- `CharArrayList.cpp`
- `unit_tests.h`
- `timer_main.cpp`
- `Makefile`
- `simple_exception.cpp`

You will be editing the first three files listed—you **should not** edit `timer_main.cpp`, `Makefile`, and `simple_exception.cpp`.
Note: The array list files just contain starter header comments; the testing file contains some examples to help you get started with testing. You will need to program everything else yourself.

You should fill out the header comments appropriately (with your name, a statement of purpose in your own words, etc.), and, of course, you will need to put the C++ code in!

Implement the array list using a dynamically allocated array as discussed in class. You should utilize a private `expand` or `ensureCapacity` function that increases/expands your capacity by a reasonable factor (and by enough to hold the required data!) each time your array list reaches its capacity.

The file `CharArrayList.h` will contain your class definition only. Put the implementation in `CharArrayList.cpp`. The file `unit_tests.h` will contain your unit tests. We will assess the work in all three files.

We may not cover exceptions in class. Don't worry! You can look up how to throw exceptions, which is all that is required for this assignment. Be sure to read `simple_exception.cpp`, which came with the starter code. That file has a few examples of how an exception can be thrown and shows how exceptions impact program execution. You should play around editing the file until you feel comfortable with exceptions. For this assignment, your exceptions should include a string with a relevant error message (which we've specified above). You'll use the `std::runtime_error` or `std::range_error` exception type, depending on the circumstance.

You will need to write unit tests to test your code. The advice section (below) has more about what kinds of things to include here; also, see the provided `unit_tests.h` for some examples.

timer_main.cpp

As mentioned above, when you copy the starter files, you will receive a file named `timer_main.cpp`. This file has been fully implemented for you. It contains a `main()` function, and when compiled and linked to your *completed* `CharArrayList` class, an excutable will be created which runs a number of `CharArrayList` operations and prints out the times (in nanoseconds) taken to run those operations. All of these operations are run on a fresh `CharArrayList` instance containing 1,000,000 random characters. Take a look over this file—it's okay if you don't understand all of it, but you may find it interesting.

Once you have **fully** completed your `CharArrayList` implementation, you can compile this timer program by running "`make timer`" in terminal. This will run the compilation rules we have defined in the `Makefile`, and create a new executable which you can run using `./timer`. Observe the table of operations and their time measurements that gets printed. Do these times match your expectations? We will ask more questions about these times below, to be answered in your `README`.

Implementation Advice

Do NOT implement everything at once!
Do NOT write code immediately!

Before writing code for any function, draw before and after pictures and write, in English, an algorithm for the function. Only after you've tried this

on several examples should you think about coding.

First, just define the class, `#include` the .h file in your `unit_tests.h` (we have already done this for you), write a dummy test (that does nothing), and run the `unit_test` command from the command line.

This will test whether your class definition is syntactically correct.

Then implement just the default constructor. Add a single variable of type `CharArrayList` to a unit test function, and run your tests.

Then you have some choices. You could add the destructor next, but certainly you should add the `toString` function soon.

You will add one function, then write code in your test file that performs one or more tests for that function. Write a function, test a function, write a function, test function, ... This is called "unit testing." As you write your functions, consider edge cases that are tricky or that your implementation might have trouble with. You should write specific tests for these cases.

Testing is an important part of programming. To give you some guidance on how to thoroughly unit test a function, we have already included some tests for the `insertAt` function in `unit_tests.h`. The contents of these tests are currently commented out—you should uncomment them after you have implemented `insertAt` (and the other functions used in those tests).

In the example tests we have given, you can see that we have considered many different cases that the `insertAt` function may be used in: insertion into an empty list, insertion at the front and back of a 1-element list, inserting a large number of elements, and inserting into the front, middle, and back of a larger list. We have even written tests that attempt to insert into and out-of-range index, then check that the correct exception was raised—you can see these for examples of how to test exceptions.

It is important to consider all cases a function can be used in when writing tests! This can help you catch bugs that may not be obvious to the human eye. You should use the `insertAt` tests we have provided as guidance for testing your other array list functions—we expect you to be similarly thorough. If you need help, the TAs will ask about your testing plan and ask to see what tests you have written.

We will evaluate your testing strategy and code for breadth (did you test all the functions?) and depth (did you identify all the normal and edge cases and test for error conditions?). Don't write a test for a function and then delete it!

In addition to testing, be sure your files have header comments, and that those header comments include your name, the assignment, the date, and the file's purpose. See our style guide for more information about commenting your code.

Finally, you should review the grading process outlined on the course administration webpage. This includes useful information like how to view your autograder score, the maximum number of times you can submit, and how many late tokens you can use.

README

With your code files you will also submit a `README` file, which you will create yourself. The file is named `README`. There is no .text or any other suffix. The contents is in plain text, lines less than 80 columns wide. Format your `README` however you like, but it should be well-organized and readable. Include the following sections:

(A) The title of the homework and the author's name (you)

(B) The purpose of the program

(C) Acknowledgements for any help you received

(D) The files that you provided and a short description of what each file is and its purpose

(E) How to compile and run your program

(F) An outline of the data structures and algorithms that you used. Given that this is a data structures class, you always need to discuss the data structure that you used and justify why you used it. Specifically for this assignment please discuss the features of array lists, and major advantages and major disadvantages of utilizing an array list as you have in this assignment. The algorithm overview may not be relevant depending on the assignment.

(G) Details and an explanation of how you tested the various parts of assignment and the program as a whole. You may reference the testing files that you submitted to aid in your explanation.

(H) Please let us know approximately how many hours you spent working on this project. This should include both weeks one and two.

(I) Answer the following questions regarding the time measurements taken for the `CharArrayList` operations (see the section titled **timer_main.cpp** for instructions on taking these measurements).

1. There are three categories of operations listed (insertion, removal, and access). Within each category, list the times each operation took and rank the operations from fastest to slowest.
2. Discuss these rankings. Why were certain operations so much faster or slower than others? Which operations took approximately the same amount of time? What are the features of array lists that caused these disparities or similarities in times?

Each of the sections should be clearly delineated and begin with a section heading describing the content of the section. It is not sufficient to just write "C" as a section header: write out the section title to help the reader of your file.

Submitting Your Work

Be sure to read over the style guide before submitting to make sure you comply with all the style requirements. The command is:

```
provide comp15 hw1_ArrayLists CharArrayList.h \
                    CharArrayList.cpp \
                    unit_tests.h \
                    Makefile
                    README
```

Note: All the file names can go on one line. The '\'s are there because we put the file names over multiple lines. In the Unix shell, a way to make a command be treated as one line when you type it on multiple lines is to "escape" the new line by putting a '\' as the last character before the new line. You may leave out the '\' and put all the file names on one terminal line, or you can do the above **exactly**, with no space after the '\'.