Programming Assignment #1: Array Lists

COP 3502, Summer 2013

Due: Friday, May 31, 11:59 PM on Webcourses2@UCF

Abstract

In this programming assignment, you will implement array lists (arrays that expand automatically whenever they get too full).

By completing this assignment, you will gain advanced experience working with dynamic memory management in C. You will also learn to manage programs that use multiple source files. In the end, you will have an awesome and useful data structure that you can reuse in the future.

Attachments

ArrayList.h, sample-main1.c, sample-main2.c, sample-output1.c, sample-output2.c, names.txt

Deliverables

ArrayList.c

1. Overview

An array list is an array that grows to accommodate new elements whenever it gets too full. As with normal arrays in C, we have direct access to any index of the array list at any given time. There are two main advantages to using array lists, though:

First, we do not need to specify the length of the array list ahead of time. This is great when we don't know ahead of time just how much data we're going to end up holding in memory. Second, we will use `get()` and `put()` functions to access individual elements of the array, and these functions will check to make sure we aren't accessing array positions that are out of bounds. (Recall that C doesn't check whether an array index is out of bounds before accessing it during program execution. That can lead to all kinds of whacky trouble!)

While Java offers native support for array lists, C does not. That's where you come in. You will implement basic array list functionality in C, including:

- Adding new elements to the end of the array list
- Automatically expanding the array list's capacity when it gets full
- Providing safe access to elements at specific positions in the array list
- Signaling to the user (i.e., the programmer (re-)using your code) when he or she attempts to access an index in the array list that is out of bounds
- Inserting elements into arbitrary positions in the array list
- … and more!

In this assignment, your array lists will be designed to hold arrays of strings. A complete list of the functions you must implement, including their functional prototypes, is given below in Section 3, "Function Requirements"). You will submit a single source file, named `ArrayList.c`, that contains all required function definitions, as well as any auxiliary functions you deem necessary. In `ArrayList.c`, you should #include any header files necessary for your functions to work, including `ArrayList.h` (see Section 2, "ArrayList.h").

Note that you will *not* write a main() function in the source file you submit! Rather, we will compile your source file with our own `main()` function(s) in order to test your code. We have included example source files that include `main()` functions, which you can use to test your code. We realize this is completely new territory for most of you, so don't panic. We've included instructions on compiling multiple source files into a single executable (e.g., mixing your `ArrayList.c` with our `ArrayList.h` and `sample-main1.c`) in Sections 4 and 5 ("Compilation and Testing").

Although we have included sample `main()` functions to get you started with testing the functionality of your code, we encourage you to develop your own test cases, as well. Ours are by no means comprehensive. We will use much more elaborate test cases when grading your submission.

Good luck!

Start early!

Come to office hours for help if you get stuck!

2. ArrayList.h

This header file contains the struct definition and functional prototypes for the array list functions you will be implementing. You should #include this file from `ArrayList.c`, like so:

```
#include "ArrayList.h"
```

The “quotes” (as opposed to <brackets>), indicate to the compiler that this header file is found in the same directory as your source, not a system directory.

You should not modify `ArrayList.h` in any way, and you should not send `ArrayList.h` when you submit your assignment. We will use our own unmodified copy of `ArrayList.h` when compiling your program.

If you write auxiliary functions in `ArrayList.c` (which is strongly encouraged!), you should ***not*** add those functional prototypes to `ArrayList.h`. Just put those functional prototypes at the top of your `ArrayList.c`.

Think of ArrayList.h as a public interface to the ArrayList data structure. It contains *only* the functions that the end user (i.e., the programmer (re-)using your code) should call in order to create and use an ArrayList. You do not want the end user to call your auxiliary functions directly, so you do not put those functional prototypes in `ArrayList.h`. That way, the end user doesn’t need to worry about all your auxiliary functions in order to use an ArrayList; everything just works. (And *you* don’t have to worry about the end user mucking everything up by accidentally calling auxiliary functions that he or she shouldn’t be messing around with!)

The basic struct you will use to implement the array lists (defined in `ArrayList.h`) is as follows:

```
typedef struct ArrayList
{
   char **array;   // pointer to array of strings
   int size;       // number of elements in array
   int capacity;   // length of array (maximum capacity)
} ArrayList;
```

The ArrayList struct contains a double `char` pointer that can be used to set up a 2D `char` array (which is just an array of `char` arrays, otherwise known as an array of strings). `array` will have to be allocated dynamically at runtime. It will probably be the bane of your existence for the next week or so.

The struct also has `size` and `capacity` variables, which store the number of elements in the array (initially zero) and the current length (i.e., maximum capacity) of the array, respectively.

3. Function Requirements

In the source file you submit, `ArrayList.c`, you must implement the following functions. You may implement any auxiliary functions you need to make these work, as well. Please be sure the spelling, capitalization, and return types of your functions match these prototypes exactly. In this section, I often refer to `malloc()`, but you're welcome to use `calloc()` or `realloc()` instead, as you see fit.

```
ArrayList *createArrayList(int length);
```

Description: Dynamically allocate space for a new ArrayList. Initialize its internal array to be of length `length` or `DEFAULT_INIT_LEN`, whichever is greater. (`DEFAULT_INIT_LEN` is defined in ArrayList.h.) Properly initialize pointers in the array to `NULL`, and set the `size` and `capacity` members of the list.

Output: "`-> Created new ArrayList of size <N>.`" (Output should not include the quotes. Terminate the line with a newline character, '`\n`'. `<N>` should of course be the length of the new array, without the angled brackets.)

Returns: A pointer to the new ArrayList, or `NULL` if any calls to `malloc()` failed.

```
ArrayList *destroyArrayList(ArrayList *list);
```

Description: Free any dynamically allocated memory associated with the list and return `NULL`.

Returns: `NULL` pointer.

```
ArrayList *expandArrayList(ArrayList *list, int length);
```

Description: Dynamically allocate a new array of length `length`. Copy the contents of `list`'s old array into the new array. Free any memory associated with the old `list→array` that is no longer in use, then set `list→array` to point to the newly created array. Be sure all pointers are properly initialized. Update the `size` and `capacity` of the ArrayList (if applicable).

Note: If `length` is less than or equal to `list`'s current array capacity, or if the `list` pointer is `NULL`, you should NOT modify the ArrayList at all. In that case, just return from the function right away without producing any output.

Output: "`-> Expanded ArrayList to size <N>.`" (Output should not include the quotes. Terminate the line with a newline character, '`\n`'. `<N>` should be the new length of the array, without the angled brackets. Do NOT produce any output if you the array is not expanded.)

Returns: A pointer to the ArrayList, or `NULL` if any calls to `malloc()` failed.

```
ArrayList *trimArrayList(ArrayList *list);
```

Description: If `list`'s `capacity` is greater than its current `size`, trim the length of the array to the current `size`. You will probably want to `malloc()` a new array to achieve this. If so, avoid memory leaks as you get rid of the old array. Update any members of `list` that need to be updated as a result of this action.

Output: "`-> Trimmed ArrayList to size <N>.`" (Output should not include the quotes. Terminate the line with a newline character, '`\n`'. `<N>` should be the new length of the array, without the angled brackets. Do NOT produce any output if the length of the array is not reduced by this function.)

Returns: A pointer to the ArrayList, or `NULL` if `malloc()` failed or the `list` pointer was `NULL`.

```
char *put(ArrayList *list, char *str);
```

Description: Insert a *copy* of `str` into the next unused cell of the array. If the list is already full, call `expandArrayList()` to grow the array to length `(capacity * 2 + 1)` before inserting the new element. When copying `str` into the array, only allocate the minimum amount of space necessary to store the string.

Returns: A pointer to the *copy* of the new string that was inserted into the array, or `NULL` if the string could not be added to the list (e.g., `malloc()` failed, or `list` or `str` was `NULL`).

```
char *get(ArrayList *list, int index);
```

Description: Attempts to return the element at the specified index. This is where you protect the user from going out-of-bounds with the array.

Returns: A pointer to the string at position `index` of the array, or `NULL` if `index` was out of bounds or the `list` pointer was `NULL`.

```
char *set(ArrayList *list, int index, char *str);
```

Description: If the array already has a valid string at position `index`, replace it with a *copy* of `str`. Otherwise, the operation fails and we simply return `NULL`. Ensure that no more space is used to store the new copy of `str` than is absolutely necessary (so, you might have to use `malloc()` and `free()` here).

Returns: A pointer to the copy of the string placed in the ArrayList, or `NULL` if the operation failed for any reason (e.g., invalid `index`, or `list` or `str` was `NULL`).

```
char *insertElement(ArrayList *list, int index, char *str);
```

Description: Insert a *copy* of `str` at the specified `index` in the array. Any elements to the right of `index` are shifted one space to the right. If the specified `index` is greater than the ArrayList's `size`, the element being inserted should be placed in the first empty position in the list.

As with the `put()` function, if the ArrayList is already full, call `expandArrayList()` to grow the array to length `(capacity * 2 + 1)` before inserting the new element. When copying `str` into the array, only allocate the minimum amount of space necessary to store the string.

Returns: A pointer to the copy of the string inserted into the list, or `NULL` if insertion fails for any reason (e.g., `malloc()` failed, or `list` or `str` was `NULL`).

```
int removeElement(ArrayList *list, int index);
```

Description: Remove the string at the specified index in the array. Strings to the right of `index` are shifted one space to the left, so as not to leave a gap in the array. The ArrayList's `size` member should be updated accordingly. If `index` exceeds the ArrayList's `size`, nothing is removed from the list.

Returns: 1 if an element was successfully removed from the array, 0 otherwise (including the case where the `list` pointer is `NULL`).

```
int getSize(ArrayList *list);
```

Description: This function returns the number of elements currently in the list. We provide this function to discourage the programmer from accessing `list→size` directly. That way, if we decide to change the name or meaning of the `size` variable in our ArrayList struct, the programmers who download the latest version of our code can get it working right out of the box; they don't have to go through their own code and change all instances of `list→size` to something else, as long as we provide them with a `getSize()` function that works.

Returns: Number of elements currently in the list, or -1 if `list` pointer is `NULL`.

```
void printArrayList(ArrayList *list);
```

Description: Print all strings currently in the array.

Output: Print all strings currently in the array. Print a newline character, '`\n`', after each string. If the list pointer is `NULL`, or if the list is empty, simply print "`(empty list)`" (without quotes), followed by a newline character, '`\n`'.

4. Compilation and Testing (CodeBlocks)

The key to getting a multiple files to compile into a single program in CodeBlocks (or any IDE) is to create a project. Here are the step-by-step instructions for creating a project in CodeBlocks, importing `ArrayList.h`, `sample-main1.c`, and the `ArrayList.c` file you've created (even if it's just an empty file so far).

1. Start CodeBlocks.

2. Create a New Project (*File -> New -> Project*).

3. Choose "Empty Project" and click "Go."

4. In the Project Wizard that opens, click "Next."

5. Input a title for your project (e.g., "ArrayList").

6. Choose a folder (e.g., Desktop) where CodeBlocks can create a subdirectory for the project.

7. Click "Finish."

Now you need to import your files. You have two options:

1. Drag your source and header files into CodeBlocks. Then right click the tab for **each** file and choose "Add file to active project."

   – or –

2. Go to *Project -> Add Files....* Browse to the directory with the source and header files you want to import. Select the files from the list (using CTRL-click to select multiple files). Click "Open." In the dialog box that pops up, click "OK."

You should now be good to go. Try to build and run the project (F9).

Note that if you import both `sample-main1.c` *and* `sample-main2.c`, the compiler will complain that you have multiple definitions for main(). You can only have one of those in there at a time. You'll have to swap them out as you test your code.

Even if you develop your code with CodeBlocks on Windows, you ultimately have to transfer it to the Eustis server to compile and test it there. See the following page (Section 5, "Compilation and Testing (Linux/Mac Command Line)") for instructions on command line compilation in Linux.

5. Compilation and Testing (Linux/Mac Command Line)

To compile multiple source files (`.c` files) at the command line:

```
gcc ArrayList.c sample-main1.c
```

By default, this will produce an executable file called `a.out` that you can run by typing:

```
./a.out
```

If you want to name the executable something else, use:

```
gcc ArrayList.c sample-main1.c -o ArrayList.exe
```

...and then run the program using:

```
./ArrayList.exe
```

Running the program could potentially dump a lot of output to the screen. If you want to redirect your output to a text file in Linux, it's easy. Just run the program using the following:

```
./ArrayList.exe > whatever.txt
```

This will create a file called `whatever.txt` that contains the output from your program.

Linux has a helpful command called `diff` for comparing the contents of two files, which is really helpful here since we've provided a sample output file. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt sample-output1.txt
```

If the contents of `whatever.txt` and `sample-output1.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
seansz@eustis:~$ diff whatever.txt solution.txt
seansz@eustis:~$ _
```

If the files differ, it will spit out some information about the lines that aren't the same. For example:

```
seansz@eustis:~$ diff whatever.txt solution.txt
6c6
< Size of list: 0
---
> Size of list: -1
seansz@eustis:~$ _
```

6. Deliverables

Submit a single source file, named `ArrayList.c`. The source file should contain definitions for all the required functions (listed above), as well as any auxiliary functions you need to make them work.

Your source file **must not** contain a `main()` function. Do not submit additional source files, and do not submit a modified `ArrayList.h` header file.

Be sure to include your name and PID as a comment at the top of your source file.

7. Grading

The expected scoring breakdown for this programming assignment is:

50%	Correct Output for Test Cases (Autograder)
40%	Implementation Details (Manually Graded)
10%	Comments and Whitespace

Your program must compile and run on Eustis to receive credit. Programs that do not compile will receive an automatic zero.

Your grade will be based largely on your program's ability to compile and produce the *exact* output expected. Even minor deviations (such as capitalization or punctuation errors) in your output will cause your program's output to marked as incorrect, resulting in severe point deductions. The same is true of how you name your functions and their parameters. Please be sure to follow all requirements carefully and test your program throughly.

Additional points will be awarded for style (proper commenting and whitespace) and adherence to implementation requirements. For example, the graders might inspect your `deleteArrayList()` function to see that it is actually freeing up memory properly.

Your `ArrayList.c` **must NOT** include a `main()` function. If it does, the autograder will fail to compile your program, and you will not receive credit for the assignment.