Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Summer 2019

Lab 8 - Developing a List Collection, Second Iteration

Objective

In this lab, you'll continue the development of a C module that implements a list collection. This lab provides a comprehensive review of structures, pointers to structures, dynamically allocated arrays and pointers to arrays.

Attendance/Demo

After you finish the exercises, a TA will review your solutions, ask you to run the test harness provided on cuLearn, and assign a grade. For those who don't finish early, a TA will grade the work you've completed, starting about 30 minutes before the end of the lab period. **Any unfinished exercises should be treated as "homework"; complete these on your own time, before your next lab.**

General Requirements

You have been provided with four files:

- array_list.h contains declarations (function prototypes) for the functions you implemented in Lab 6 plus the ones you'll implement in this lab. **Do not modify array_list.h.**
- additional_functions.c contains incomplete definitions of several functions you have to design and code;
- main.c and sput.h implement a *test harness* (functions that will test your code, and a `main` function that calls these test functions). **Do not modify `main()` or any of the test functions.**

You will also need the array_list.c file that you implemented during Lab 7.

None of the functions you write should call `calloc`, `realloc` or `free`. Only `list_construct` (from Lab 7) and `increase_capacity` (Exercise 4) are permitted to call `malloc`.

None of the functions you write should perform console input; i.e., contain `scanf` statements. Unless otherwise specified, none of your functions should produce console output; i.e., contain `printf` statements.

You must format your C code so that it adheres to one of two commonly-used conventions for indenting blocks of code and placing braces (K&R style or BSD/Allman style). Instructions for selecting the formatting style and formatting blocks of code are in the Lab 1 handout.

Finish each exercise (i.e., write the function and verify that it passes all its tests) before you

move on to the next one. Don't leave testing until after you've written all your functions.

Getting Started

Step 1: Launch Pelles C and create a new project named array_list_v2. (Instructions for creating projects are in the handout for Lab 1.) If you're using the 64-bit edition of Pelles C, select Win 64 Console program (EXE) as the project type. If you're using the 32-bit edition of Pelles C, select Win32 Console program (EXE). **Don't click the icons for Console application wizard, Win32 Program (EXE) or Win64 Program (EXE) - these are not correct types for this project.**

Step 2: Download file main.c, additional_functions.c, array_list.h and sput.h from cuLearn. Move these files into your array_list_v2 folder.

Step 3: Move a copy of the array_list.c file that you developed in Lab 7 into your array_list_v2 folder. **Do not put copies of Lab 7's array_list.h or main.c (the test harness) in this folder.** You've been provided with a new versions of these files for this week's lab (see the previous step).

Step 4: Add main.c and array_list.c to your project. (Instructions for doing this are in the handout for Lab 1.) **Do not add additional_functions.c to your project.**

You don't need to add array_list.h and sput.h to the project. Pelles C will do this after you've added main.c.

Step 5: In this lab, you're going to implement several new functions in array_list.c. File additional_functions.c contains incomplete definitions of these functions, which you need to copy into array_list.c. To do this:

- open array_list.c and additional_functions.c.
- select and copy all the function definitions (including the header comments) in additional_functions.c and paste them at the end of array_list.c.
- close additional_functions.c.

Step 6: Build the project. It should build without any compilation or linking errors.

Step 7: The test harness contains test suites for the functions from Lab 7, as well as test suites for the functions you'll write this week. As we incrementally develop a module, it's important to retest all the functions, to ensure that the changes we make don't "break" functions that previously passed their tests. This testing technique is known as *regression testing*.

Execute the project. Test suites #1 through #7 should pass. If these suites report errors, there are flaws in the code you wrote last week, which you'll need to correct before you work on this week's exercises. Tests suite #8 will report errors, which is what we'd expect, because you haven't started working on the function this suite tests.

Step 8: Do Exercises 1 through 5. There is an extra-practice exercise (Exercise 6) at the end of the handout, but this exercise will not be graded during the lab.

Exercise 1

File array_list.c contains an incomplete definition of a function named `list_index`. This function returns the index (position) of the first occurrence of a specified "target" integer in a list. If the integer is not in the list, the function should return -1.

The function prototype is:

```
int list_index(const list_t *list, int target)
```

This function should terminate (via `assert`) if parameter `list` is `NULL`.

Complete the function definition.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Use the console output to help you identify and correct any flaws. Verify that `list_index` passes all the tests in test suite #8.

Exercise 2

File array_list.c contains an incomplete definition of a function named `list_count`. This function counts the number of times that a specified "target" integer occurs in a list, and returns that number. The function prototype is:

```
int list_count(const list_t *list, int target);
```

This function should terminate (via `assert`) if parameter `list` is `NULL`.

Complete the function definition.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Use the console output to help you identify and correct any flaws. Verify that `list_count` passes all the tests in test suite #9.

Exercise 3

File array_list.c contains an incomplete definition of a function named `list_contains`. This function returns `true` if a list contains a specified "target" integer; otherwise it should return `false`. The function prototype is:

```
_Bool list_contains(const list_t *list, int target);
```

This function should terminate (via `assert`) if parameter `list` is `NULL`.

Complete the function definition. Don't write the function body "from scratch". Instead, this function must call one of the other functions is your module. Pick the function that results in the most efficient implementation of `list_contains`.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Use the console output to help you identify and correct any flaws. Verify that `list_contains` passes all the tests in test suite #10.

Exercise 4

Lists created by `list_construct` have a fixed capacity, and if the `list_append` function you implemented in Lab 7 is passed a pointer to a list that is full, it returns without modifying the list. We would like to remove this limitation.

File array_list.c contains an incomplete definition of a function named `increase_capacity` that enlarges a list's capacity to a new capacity. Here is the function prototype:

```
void increase_capacity(list_t *list, int new_capacity);
```

This function should terminate (via `assert`) if the new capacity is not greater than the list's current capacity or if memory for the backing array cannot be allocated.

The function should not change the order of the integers stored in this list. For example, suppose `increase_capacity` is passed a pointer to a full list that contains `[4 7 3 -2 9]`. The second argument (the new capacity) is `20`. When the function returns, the list's capacity will have been increased to `20`, and the order of the integers it contains are unchanged (`4` is stored at index `0`, `7` is stored at index `1`, etc.)

Complete the function definition.

Notes:

- It's not enough to change the value stored in the `list_t` structure's `capacity` member to the specified new value. That won't change the list's capacity. To increase the capacity, the function must replace the list's backing array with a larger one.
- Your function must call `malloc` to allocate the new backing array for the list. Do not use C's `realloc` or `calloc` functions.
- Before it returns, your function must free any heap memory that is no longer used by the list.
- When designing this function, it may help to draw some memory diagrams that show the step-by-step changes that will happen to the heap (that is, the `list_t` struct and the list's backing array) as its capacity is increased.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Use the console output to help you identify and correct any flaws. Verify that `increase_capacity` passes all the tests in test suite #11.

Exercise 5

Edit the header comment for `list_append` so that it looks like this:

```
/* Insert element at the end of the list pointed to by parameter list
 * and return true.
 * If the list is full, double its capacity before appending
 * the element.
 * Terminate the program via assert if list is NULL.
 * Terminate the program via assert if the full list's capacity
 * couldn't be doubled.
 */
```

Modify your `list_append` function so that, if the list is full, it doubles the list's capacity before appending the integer to the list. The function's return type and parameter list must not be changed. Your function must call your `increase_capacity` function.

Unlike the first implementation of `list_append`, this version of the function never returns `false`. It will always return `true` or terminate via `assert`. Make sure you understand why.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Use the console output to help you identify and correct any flaws. Verify that `list_append` passes all the tests in test suite #12.

Wrap-up

1. Remember to have a TA review your solutions to the exercises, assign a grade (Satisfactory, Marginal or Unsatisfactory) and have you initial the grading/signout sheet sheet.

2. Remember to backup your project folder before you leave the lab; for example, copy it to a flash drive and/or a cloud-based file storage service. All files you've created on the hard disk will be deleted when you log out.

Extra Programming Practice (**Exercise 6)**

File array_list.c contains an incomplete definition of a function named `list_delete`. This function deletes the integer at a specified position in a list. The function prototype is:

```
void list_delete(list_t *list, int index);
```

Parameter `index` is the index (position) of the integer that should be removed. If a list contains `size` integers, valid indices range from `0` to `size-1`.

This function should terminate (via `assert`) if parameter `list` is `NULL` or if parameter `index` is not valid.

When your function deletes the integer at position `index`, the array elements at positions `0`

through `index`-1 will not change; however, the elements at positions `index`+1 through `size`-1 must all be "shifted" one position to the left. Example: if a list contains `[2 4 6 8 10]`, then calling `list_delete` with `index` equal to `2` deletes the `6` at that position, changing the list to `[2 4 8 10]`. Notice that `8` has been copied from position `3` to position `2`, and `10` has been copied from position `4` to position `3`.

Complete the function definition. **Do not use `malloc` to allocate a new backing array. Do not declare an array inside your function; that is, your function cannot have a declaration of the form `int a[n];`**

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Use the console output to help you identify and correct any flaws. Verify that `list_delete` passes all the tests in test suite #13.

Homework Exercise - Visualizing Program Execution

In the final exam, you will be expected to be able to draw diagrams that depict the execution of short C programs that use pointers to dynamically-allocated structs and arrays, using the same notation as C Tutor. This exercise is intended to help you develop your code tracing/visualization skills when working with programs that allocate allocate memory from the heap.

1. The *Labs* section on cuLearn has a link, Open C Tutor in a new window. Click on this link.

2. Copy the `list_t` declaration from `array_list.h`, your solutions to Exercises 1-5, plus `list_construct` and any functions from Lab 7 that are called by the functions you wrote for this lab, into C Tutor.

3. Write a short `main` function that exercises your list functions.

4. *Without using C Tutor,* trace the execution of your program. Draw memory diagrams that depict the program's activation frames and the heap just before each of your list functions returns. Use the same notation as C Tutor.

5. Use C Tutor to trace your program one statement at a time, stopping just before each function returns. Compare your diagrams to the visualizations displayed by C Tutor.