Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Winter 2015

Lab 8 - Developing a List Collection, Second Iteration

Objective

To 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

To receive credit for this lab, you must demonstrate your work. **Also, you must submit your work to cuLearn by the end of the lab period**. (Instructions are provided in the *Wrap Up* section at the end of this handout.)

When you have finished all the exercises, call a TA, who will review the code you wrote. For those who don't finish early, a TA will ask you to demonstrate whatever code 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

Finish each exercise (i.e., write the function and verify that it passes all of its tests) before you move on to the next one. Don't leave testing until after you've written all your functions.

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 have been provided with four files:

- additional_functions.c contains incomplete definitions of several functions you have to design and code;
- additional_prototypes.h contains declarations (function prototypes) for the functions you'll implement.
- 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.**

Instructions

1. Create a new folder named Lab 8.

2. Launch Pelles C and create a new Pelles C project named array_list_v2 inside your Lab 8 folder. The project type must be Win32 Console program (EXE). You should now have a folder named array_list_v2 inside your Lab 8 folder. Check this. If you do not have a project folder named array_list_v2, close this project and repeat Step 2.

3. Copy your array_list.c and array_list.h files from Lab 7 into your array_list_v2 folder. **Do not copy main.c (the test harness) from Lab 7.** You've been provided with a new main.c for this week's lab.

4. Download file main.c, additional_functions.c, additional_prototypes.h and sput.h from cuLearn. Move these files into your array_list_v2 folder.

5. You must also add main.c and array_list.c to your project. From the menu bar, select Project > Add files to project... In the dialogue box, select main.c, then click Open. An icon labelled main.c will appear in the Pelles C project window. Repeat this for array_list.c.

   Do not add additional_functions.c and additional_prototypes.h 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. and array_list.c.

6. In this lab, you're going to implement several additional functions in the module you developed in Lab 7. To get started, you need to add the declarations (prototypes) for these functions to array_list.h, and incomplete implementations to array_list.c.

   6.1. Open array_list.h and additional_prototypes.h. Copy all the function prototypes in additional_prototypes.h and paste them at the end of array_list.h. Close additional_prototypes.h.

   6.2. Open array_list.c and additional_functions.c. Copy all the function definitions in additional_prototypes.c and paste them at the end of array_list.c. Close additional_functions.h.

7. Build the project. It should build without any compilation or linking errors.

8. 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. When we incrementally develop a module, its important to retest all of 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 they don't, there are problems with the code you wrote last week. You'll need to fix these flaws before you work on this week's exercises). Tests suites #8 through #14 will report several errors, which is what we'd expect, because you haven't started working on the functions these

suites test.

Exercise 1

File array_list.c contains an incomplete definition of a function named `intlist_index`. This function returns the index (position) of the first occurrence of an integer in the list pointed to by parameter `list`. The function prototype is:

```
int intlist_index(const IntList *list, int target);
```

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

If `target` is in the list, the function should return the index of the first occurrence. If `target` is not in the list, the function should return -1.

Finish the implementation of this function.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Inspect the console output, and verify that your `intlist_index` function passes all the tests in test suite #8 before you start Exercise 2.

Exercise 2

File array_list.c contains an incomplete definition of a function named `intlist_count`. This function counts the number of occurrences of a specified integer in the list pointed to by parameter `list`, and returns that number. The function prototype is:

```
int intlist_count(const IntList *list, int target);
```

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

The function returns the count of the number of times that `target` is found in the list.

Finish the implementation of this function.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Inspect the console output, and verify that your `intlist_count` function passes all the tests in test suite #9 before you start Exercise 3.

Exercise 3

File array_list.c contains an incomplete definition of a function named `intlist_contains`. This function determines if the list pointed to by parameter `list` contains a specified integer. The function prototype is:

```
_Bool intlist_contains(const IntList *list, int target);
```

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

If `target` is in the list, the function should return `true`; otherwise it should return `false`.

Finish the implementation of this function.

Hint: you can implement this function in only few lines of code by calling one or more of the other functions in your module.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Inspect the console output, and verify that your `intlist_contains` function passes all the tests in test suite #10 before you start Exercise 4.

Exercise 4

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

```
    void intlist_delete(IntList *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 `intlist_delete` with `index` equal to 2 changes 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.

Finish the implementation of this function.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Inspect the console output, and verify that your `intlist_delete` function passes all the tests in test suite #11 before you start Exercise 5.

Exercise 5

File array_list.c contains an incomplete definition of a function named `intlist_remove`.

This function removes the first occurrence of a specified integer in the list pointed to by parameter `list`. The function prototype is:

```
    _Bool intlist_remove(IntList *list, int target);
```

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

If `target` is found and removed from the list, this function should return `true`. If `target` is not found, the function should leave the list unchanged and return `false`.

If `target` is found at position *i*, the array elements at positions 0 through *i*-1 will not change; however, the elements at positions *i*+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 `intlist_remove` with `target` equal to 8 changes the list to [2 4 6 10].

Finish the implementation of this function.

Hint: you can implement this function in only few lines of code by calling one or more of the other functions in your module.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Inspect the console output, and verify that your `intlist_remove` function passes all the tests in test suite #12 before you start Exercise 6.

Exercise 6

Lists created by `intlist_construct` have a fixed capacity, and `intlist_append` currently returns `false` if it attempts to add an integer to a full list. We would like to remove this limitation.

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

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

This function should terminate (via `assert`) if parameter `list` is `NULL` or if the new capacity is not greater than the list's current capacity.

The function should not change the order of the integers stored in this list; for example, suppose a list contains [4 7 3 -2 9] when `increase_capacity` is called. When the function returns, the list's capacity will have been increased, and it will contain the same integers, in the same order (4 is stored at index 0, 7 is stored at index 1, etc.)

Finish the implementation of this function. It's up to you to decide how the function should handle any memory allocation errors that occur while it is executing.

Hint: it's not enough to change the value stored in the `IntList` structure's `capacity` member!

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Inspect the console output, and verify that your `increase_capacity` function passes all the tests in test suite #13 before you start Exercise 7.

Exercise 7

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

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Inspect the console output, and verify that your `intlist_append` function passes all the tests in test suite #14.

Wrap-up

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

2. The next thing you'll do is package the project in a ZIP file (compressed folder) named array_list_v2.zip. To do this:

   2.1. From the menu bar, select Project > ZIP Files... A Save As dialog box will appear. If you named your Pelles C project array_list_v2, the zip file will have this name by default; otherwise, you'll have to edit the File name: field and rename the file to array_list_v2 before you save it. **Do not use any other name for your zip file** (e.g., lab8.zip, my_project.zip, etc.).

   2.2. Click Save. Pelles C will create a compressed (zipped) folder, which will contain copies of the the source code and several other files associated with the project. (The original files will not be removed). The compressed folder will be stored in your project folder (i.e., folder array_list_v2).

3. Before you leave the lab, log in to cuLearn and submit array_list_v2.zip. To do this:

   3.1. Click the Submit Lab 8 link. A page containing instructions and your submission status will be displayed. After you've read the instructions, click the Add submission button. A page containing a File submissions box will appear. Drag array_list_v2.zip to the File submissions box. **Do not submit another type of file (e.g., a Pelles C .ppj file, a RAR file, a .txt file, etc.)**

   3.2. After the icon for the file appears in the box, click the Save changes button. At this point, the submission status of your file is "Draft (not submitted)". If you're ready to finish submitting the file, jump to Step 3.4. If you instead want to replace or delete your "draft" file submission, follow the instructions in Step 3.3.

   3.3. You can replace or delete the file by clicking the Edit my submission button. The page containing the File submissions box will appear.

      3.3.1. To overwrite a file you previously submitted with a file having the same name, drag another copy of the file to the File submissions box, then click the Overwrite button when you are told the file exists ("There is already a file called..."). After the icon for the file reappears in the box, click the Save changes button.

      3.3.2. To delete a file you previously submitted, click its icon. A dialogue box will appear. Click the Delete button., then click the OK button when you are asked, "Are you sure you want to delete this file?" After the icon for the file disappears, click the Save changes button.

3.4. Once you're sure that you don't want to make any changes, click the Submit assignment button. A Submit assignment page will be displayed containing the message, "Are you sure you want to submit your work for grading? You will not be able to make any more changes." Click the Continue button to confirm that you are ready to submit your lab work. This will change the submission status to "Submitted for grading".