Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Winter 2016

Lab 8 - Developing a Dynamic Array (Array List), Second Iteration

Objective

To continue the development of a C module that implements a dynamic array (array list).

Attendance/Demo

To receive credit for this lab, you must demonstrate your solutions to the exercises. **Also, you must submit your work to cuLearn**. (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 three files:

- array_list.h contains declarations (function prototypes) for the functions you implemented in Lab 7 and the ones you'll implement in this lab. **Do not modify array_list.h.**
- 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.

Instructions

1. Launch Pelles C and create a new Pelles C project named array_list_v2 (all letters are lowercase, with underscores separating the words). The 64-bit version of Pelles C is installed on our lab computers, so the project type must be Win 64 Console program (EXE). (Although the 64-bit edition of Pelles C can build 32-bit programs, you may run into difficulties if you attempt to use the debugger to debug 32-bit programs.) If you're using your own computer, the project type should be Win 64 Console program (EXE) or Win32 Console program (EXE), depending on whether you installed the 64-bit or 32-bit edition of Pelles C. After creating the project, you should have a folder named array_list_v2. Check this. If you do not have a project folder named array_list_v2, close this project and repeat Step 1.

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

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

4. 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.

   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.

5. Open array_list.c in the editor. Put these function definitions at the end of array_list.c.

```
/* Return the index (position) of the first occurrence of an integer
 * equal to target in the specified list.
 * Return -1 if target is not in the list.
 */
int intlist_index(const intlist_t list, int target)
{
    return -2;
}

/* Count the number of integers that are equal to target in the
 * specified list, and return that number.
 */
int intlist_count(const intlist_t list, int target)
{
    return -1;
}
```

```
/* Determine if an integer in the specified list is equal to target.
 * Return true if target is in the list, otherwise return false.
 */
_Bool intlist_contains(const intlist_t list, int target)
{
    return false;
}

/* Increase the capacity of the specified list to new_capacity, and
 * return the list.
 *
 * Example: suppose a list passed to increase_capacity has capacity 5
 * and contains [1, 2, 3, 4, 5]; i.e., its size is 5. If new_capacity
 * is 10, the list returned by this function will contain
 * [1, 2, 3, 4, 5], so the list size will be 5, but its capacity will
 * be 10.
 *
 * Terminate the program via assert if new_capacity is not greater
 * than the list's current capacity.
 */
intlist_t increase_capacity(intlist_t list, int new_capacity)
{
    return list;
}

/* Delete the integer at the specified position in the specified
 * list, and return the modified list.
 * Parameter index is the position of the integer that should be
 * removed.
 * Terminate the program via assert if index is not in the range
 * 0 .. intlist_size() - 1.
 */
intlist_t intlist_delete(intlist_t list, int index)
{
    return list;
}
```

Replace the header comment for `intlist_append` with this comment:

```
/* Insert elem at the end of the specified list, and return the list.
 * If the list is full, double the list's capacity before inserting
 * the element.
 */
```

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

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 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 #13 will report several errors, which is what we'd expect, because you haven't started working on the functions these suites test.

8. Design and code the functions described in Exercises 1 through 5. There is an extra-practice exercise (Exercise 6) at the end of the lab handout.

Exercise 1 (Hint: the function body requires fewer than 10 lines of code, including lines that contain only })

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 a list. The function prototype is:

```
int intlist_index(const intlist_t list, int target);
```

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 (Hint: the function body requires fewer than 10 lines of code, including lines that contain only })

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

```
int intlist_count(const intlist_t list, int target);
```

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 (Hint: the function body requires fewer than 5 lines of code)

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

```
_Bool intlist_contains(const intlist_t list, int target);
```

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.

Exercises 4 and 5 are a bit more challenging that the first three exercises. If you don't have the time to complete these two exercises during the lab, make sure you do them before the final exam.

Exercise 4 (Hint: the function body requires fewer than 15 lines of code, including lines that contain only })

Lists created by `intlist_construct` have a fixed capacity, and if `intlist_append` is passed a when 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:

```
intlist_t increase_capacity(intlist_t list, int new_capacity);
```

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

This function will modify the `intlist_t` structure stored in parameter `list`, so it must always return this structure (for more information, see the lecture slides on dynamic arrays).

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 to the specified larger capacity, 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. Your function must call `malloc` to allocate a new

backing array for the list. **It is not permitted to call C's `realloc` function.**

Hint 1: while designing this function, before you start coding, draw some memory diagrams that show the changes that happen to an `intlist_t` structure as it's capacity is increased.

Hint 2: it's not enough to change the `intlist_t` structure's `capacity` member to the specified new capacity. That doesn't change the `intlist_t`'s capacity - to do that, the function must replace the list's backing array with a larger one.

Hint 3: before it returns, your function must free any heap memory that is no longer used by the list.

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 #11 before you start Exercise 5.

Exercise 5 (Hint: the function body requires fewer than 10 lines of code, including lines that contain only `}`)

Modify your `intlist_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.

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 #12.

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. 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 aren't ready to do this; for example, you want to do some more work on the project and resubmit it later, you can leave the file with "draft" submission status. When you're ready to submit the final version, you can replace or delete your "draft" file submission by following the instructions in Step 3.3, then finish the submission process by following the instructions in Step 3.4.

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".

Extra Practice - Exercise 6 (Hint: the function body requires fewer than 10 lines of code, including lines that contain only }))

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

```
intlist_t intlist_delete(intlist_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 `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 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.

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 #13.