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

Lab 7 - 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.

Online Submission

Submit file array_list.c through cuLearn before the deadline.

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 6.

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). Pelles C makes it easy to do this - instructions were provided in Labs 1 and 2.

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.

Getting Started

Step 1: Launch Pelles C and create a new Pelles C project named array_list_v2.

- If you're using the 64-bit edition of Pelles C, the project type should 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 the 32-bit edition of Pelles C, the project type should be Win32 Console program (EXE).

When you finish this step, Pelles C will create a folder named array_list_v2.

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: Copy the array_list.c file that you developed in Lab 6 into your array_list_v2 folder. **Do not copy array_list.h or main.c (the test harness) from Lab 6.** You've been provided with a new versions of these files for this week's lab (see the previous step).

Step 4: You must also add main.c and array_list.c to your project. To do this:

- select Project > Add files to project... from the menu bar.
- 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 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. To get started, you need to copy these definitions to 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 6, 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 #8 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 #9 through #14 will report several errors, which is what we'd expect, because you haven't started working on the functions these suites test.

Step 8: Open array_list.c in the editor and 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 `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)
```

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 #9.

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 a list and returns that number. The function prototype is:

```
int intlist_count(const intlist_t *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 #10.

Exercise 3

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);
```

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. This requires only few lines of code if it calls 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 #11.

Exercise 4

Lists created by `intlist_construct` have a fixed capacity, and if the `intlist_append` function you implemented in Lab 6 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(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 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 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. Notes:

- It's not enough to change the value stored in the `intlist_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. You are not permitted to call C's `realloc` function.
- 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 `intlist_t` structure 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. Inspect the console output, and verify that your `increase_capacity` function passes all the tests in test suite #12.

Exercise 5

Edit the header comment for `intlist_append` from Lab 6 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 the list's capacity before inserting
```

```
 * the element.
 * Terminate the program via assert if list is NULL.
 */
```

Modify your `intlist_append` function from Lab 6 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 `intlist_append` from Lab 6, 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. Inspect the console output, and verify that your `intlist_append` function passes all the tests in test suite #13.

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 attendance/grading sheet.

2. Remember to back up 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 `intlist_delete`. This function deletes the integer at a specified position in a list. The function prototype is:

```
void 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 `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 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. **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. Inspect the console output, and verify that your `intlist_delete` function passes all the tests in test suite #14.

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 `struct`s 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. Copy the `intlist_t` declaration from `array_list.h`, your solutions to Exercises 1-5, `intlist_construct` and any functions from Lab 6 that are called by the functions you wrote for this lab, into C Tutor.

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

3. *Without using C Tutor,* trace the execution of your program. Draw memory diagrams that depict the program's activation frames just before the `return` statements in each of your list functions are executed. Use the same notation as C Tutor.

4. Use C Tutor to trace your program one statement at a time, stopping just before each `return` statement is executed. Compare your diagrams to the visualizations displayed by C Tutor.