Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Fall 2013

Lab 12 - Developing an Array-Based List Collection, Second Iteration

Objective

After completing Labs 10 and 11, you should have a good understanding of the algorithms that perform many of the fundamental operations on linked list. In this lab we're going to revisit the list collection that we started in Lab 7. This lab will review structures, pointers to structures, dynamically allocated arrays and pointers to arrays.

Attendance/Demo

To receive credit for this lab, you must make the effort to finish a reasonable number of exercises and demonstrate the code you complete.

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. Finish any exercises that you don't complete by the end of the lab on your own time. Also, you must submit your lab work to cuLearn by the end of the lab period. (Instructions are provided in the *Wrap Up* section at the end of this handout.)

General Requirements

In Exercises 1 through 8, you are going to define functions that operate on fixed-capacity lists of integers. Finish each exercise (i.e., write the function that performs the specified list operation, 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 the functions.

None of the functions you define in array_list.c 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.c contains incomplete definitions of several functions you have to design and code.;
- array_list.h contains declaration (function prototypes) for the functions you'll implement. **Do not modify array_list.h unless you are told to do so.**
- test_array_list.c contains a *test harness* consisting of 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 unless you are told to do so.** Unlike the test harnesses provided in some previous labs, this one does not compare the actual and expected results of each test and keep track of the number of tests that pass and fail. Instead, results are displayed on the console, and you have to review this output to determine if your functions are correct.

Although it's not required, you may find it useful to have a copy of your intlist project from Lab 7. If you don't have a copy, you should be able to download the zip file you submitted to cuLearn and extract the project.

Instructions

1. Launch Pelles C and create a new Pelles C project named array_list. The project type must be Win32 Console program (EXE). You should now have a folder named array_list.

2. Download file test_array_list.c, array_list.c and array_list.h from cuLearn. Move these files into your array_list folder. **You must also add test_array_list.c and array_list.c to your project:** from the menu bar, select Project > Add files to project... In the dialogue box, select test_array_list.c, then click Open. An icon labelled test_array_list.c will appear in the Pelles C project window. Repeat this for array_list.c. Pelles C will automatically add array_list.h to the project.

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

4. Execute the project. The test harness will run. Read the console output carefully. Read the main function, and determine what the tests do.

Exercise 1

File array_list.h contains the declaration for a list collection that uses a dynamically allocated array as the underlying data structure. Here is the declaration:

```
struct intlist {
     int   *elems;
     int   capacity; // Maximum number of elements in the list.
     int   size;     // Current number of elements in the list.
};

typedef struct intlist IntList;
```

In array_list.c, define a function that returns a pointer to a new, empty list of integers with the specified capacity. The function prototype is:

```
int *intlist_construct(int capacity);
```

This function terminates (via `assert`) if `capacity` is less than or equal to 0.

This function must allocate two blocks of memory from the heap. One block is the dynamically allocated `IntList` structure and the other block is the dynamically allocated array with the specified capacity. The function must terminate if the required amount of memory cannot be allocated. Remember to save the pointer to the array in member `elems`, and to initialize the `capacity` and `size` members.

Run the test harness, and examine the console output to determine if your `intlist_construct` function is correct.

Exercise 2

In array_list.c, define a function that prints the integers stored in the list pointed to by parameter `list`. The function prototype is:

```
void intlist_print(const IntList *list)
```

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

The required format for the output is [$elem_0$ $elem_1$ $elem_2$ ... $elem_{n-1}$]; that is, a list of integers enclosed in square brackets, with one space between each pair of values. There must be no spaces between the '[' and the first value, or between the last value and ']'.

For example, if `intlist_print` is passed a list containing 1, 5, -3, and 9, the output produced by this function should look exactly like this:

```
[1 5 -3 9]
```

If the list is empty (length 0), the output should be: `[]`.

Hint: you wrote a similar function in Lab 7.

Run the test harness, and examine the console output to determine if your `intlist_print` function is correct.

Exercise 3

In array_list.c, define a function that appends an integer to the end of the list pointed to by parameter `list`. The function prototype is:

```
_Bool intlist_append(IntList *list, int element)
```

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

If `element` was appended, the function should return `true`. If the function was not successful, because the list was full, it should leave the list unchanged and return `false`.

Hint: you wrote a similar function in Lab 7.

Run the test harness, and examine the console output to determine if your `intlist_append` function is correct.

Exercise 4

In array_list.c, define a function that returns the capacity of the list pointed to by parameter `list`. The function prototype is:

```
int intlist_capacity(const IntList *list)
```

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

Run the test harness, and examine the console output to determine if your `intlist_capacity` function is correct.

Exercise 5

In array_list.c, define a function that returns the size of the list pointed to by parameter `list`. The function prototype is:

```
int intlist_size(const IntList *list)
```

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

Run the test harness, and examine the console output to determine if your `intlist_size` function is correct.

Exercise 6

In array_list.c, define a function that returns the element located at the specified index in the list pointed to by parameter `list`. The function prototype is:

```
int intlist_get(const IntList *list, int index)
```

This function should terminate (via `assert`) if parameter `list` is `NULL` or if `index` is not in the range `0` .. `intlist_size()-1`.

Run the test harness, and examine the console output to determine if your `intlist_get` function is correct.

Exercise 7

In array_list.c, define a function that stores the specified element at the specified index in the list pointed to by parameter `list`. The function will return the integer that was previously stored at that index.

The function prototype is:

```
int intlist_set(IntList *list, int index, int element)
```

This function should terminate (via `assert`) if parameter `list` is `NULL` or if `index` is not in the range `0` .. `intlist_size()-1`.

Run the test harness, and examine the console output to determine if your `intlist_set` function is correct.

Exercise 8

In array_list.c, define a function that empties the list pointed to parameter `list`. The function

prototype is:

```
void intlist_removeall(IntList *list)
```

This function should terminate (via `assert`) if parameter `list` is `NULL`. This function does not free any of the memory that was allocated by `intlist_construct`, so the emptied list can continue to be used.

Run the test harness, and examine the console output to determine if your `intlist_removeall` function is correct.

Wrap-up

1. Remember to have a TA review and grade your solutions to the exercises before you leave the lab.

2. The next thing you'll do is package the project in a ZIP file (compressed folder). From the menu bar, select Project > ZIP Files... A Save As dialog box will appear. Click Save. Pelles C will create a compressed (zipped) folder named array_list.zip, 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).

3. Log in to cuLearn, click the Submit Lab 12 link and submit array_list.zip. After you click the Add submission button, drag the file to the File submissions box. After the icon for the file appears in the box, click the Save changes button. At this point, the submission status for your file is "Draft (not submitted)". You can resubmit the file by clicking the Edit my submission button. After you've finished uploading your file, remember to click the Submit assignment button. This will change the submission status to "Submitted for grading".

See the following pages for some extra exercises to help you prepare for the final exam.

Extra Exercises

Exercise 9

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.

In array_list.c, define a function named `increase_capacity` that attempts to enlarge a list's capacity to the specified value. Here is the function prototype:

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

You can assume that parameter `new_capacity` is greater than the list's current capacity (your function does not have to check this). 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.) It's up to you to decide how the function should handle any errors that occur while it is executing.

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

Exercise 10

Modify your `intlist_append` function so that, if the list is full, 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 .

In test_array_list.c, modify `test_intlist_append` to verify that your modified `intlist_append` function doubles the capacity of a full list.

For Exercises 11 through 15, you should be able to reuse code you wrote for Lab 7. You will need to define additional test functions to test_array_list.c.

File array_list.h contains commented-out prototypes for the functions you'll write in Exercises 11 through 15. Edit array_list.h and remove the comments from around these declarations.

Exercise 11

In array_list.c, define a function that returns the index (position) of the first occurrence of an integer equal to `target` 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 not in the list, the function should return -1.

Exercise 12

In array_list.c, define a function that counts the number of integers in the list pointed to by parameter `list` that are equal to `target`, 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`.

Exercise 13

In array_list.c, define a function that determines if an integer in the list pointed to by parameter `list` is equal to `target`. The function prototype is:

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

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

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

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

Exercise 14

In array_list.c, define a function that deletes the integer at the specified position in the list pointed to by parameter `list`. The function prototype is:

```
_Bool 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`.

If the integer was deleted, this function should return `true`. If the function was not successful, because the list was empty or because parameter `index` is not a valid, it should leave the list unchanged and return `false`.

Hint: 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 `len`-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.

Exercise 15

In array_list.c, define a function that removes the first occurrence of an integer equal to `target` 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, because it is not in the list or the list is empty, the function should leave the list unchanged and return `false`.

Hint: 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 `len`-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].