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

Lab 7 - Developing a List Collection, First Iteration

Objective

To begin the development of a C module that implements a list collection.

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.**

Background

C (and C++) arrays have several limitations:

- An array's capacity is specified when the array is declared. This capacity is fixed, so there is no way to increase the array's capacity at run-time. Also, C does not have an operator or standard library function that determines an array's capacity.

- C does not keep track of how many array elements have been initialized. It is the programmer's responsibility to do this. For example, here is the declaration for an array with capacity 10, named `a`:

  ```
  int a[10];
  ```

  This loop stores integers 1 through 5 in the first five array elements:

  ```
  for (int i = 0; i < 5; i = i + 1) {
      a[i] = i + 1;
  }
  ```

  There is no way to "ask" this array whether or not a particular element has been initialized.

- C does not check for out-of-bounds array indices, which means code can access memory outside the array by using an out-of-bounds array index. The expressions `a[-1]` or `a[10]` will compile without error (even though the declared capacity of the array is 10). At run-time, these expressions will not cause the program to terminate with an error, even though they access memory outside of array `a`.

Many modern programming languages have addressed these limitations by providing a collection known as a *list*. For example, Java provides a class named `ArrayList` and Python has a built-in class named `list`. Although C++ supports C-style arrays for backwards compatibility, many C++ programmers instead use the `vector` class that is part of the C++ Standard Template Library.

Here are the important differences between arrays and the lists provided by many programming languages:

- A list increases its capacity as required. As you append items to a list or insert items in a list, the list will automatically grow (increase its capacity) when it becomes full. For this reason, a list is sometimes thought of as a *dynamic array*.

- A list keeps track of its *length* or *size* (that is, the number of items currently stored in the list). Python has a built-in `len` function that takes one argument, a list, and returns the list's length. Java's `ArrayList` class provides a *method* (another name for a function) named `size`, which returns the number of items in the list.

- Lists will often generate a run-time error if you specify an out-of-range list index. By default, this normally results in an error message being displayed, then the program terminates.

- In Python, many common list operations are provided by built-in operators, functions and methods. Java's `ArrayList` class defines several methods that provide similar list operations. Compare this with C and C++ arrays - there are very few built-in array operations.

Over the next couple of labs, you're going to develop a C module that implements a list collection. This collection will provide many of the same features as Python's `list`, Java's `ArrayList` and C++'s `vector`, and will be a useful module to have in your "toolbox" if you end up doing a lot of C programming.

In the first version of this module, we won't attempt to implement all the features of Python or Java lists. Although our list will be based on a dynamically-allocated array, in this first iteration it will have fixed capacity; in other words, it won't grow when it becomes full. We're going to focus on developing functions that provide some common list operations. You'll refine and extend your module in subsequent labs.

We'll use the following terms when working with lists:

- list *length*: the number of items currently stored in a list

- list *size*: a synonym for length

- list *capacity*: the maximum number of items that can be stored in a list

Make sure you understand the difference between a list's length (size) and its capacity.

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:

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

Getting Started

1. Create a new folder named Lab 7.
2. Launch Pelles C and create a new Pelles C project named array_list inside your Lab 7 folder. The project type must be Win32 Console program (EXE). After creating the project, you should now have a folder named array_list inside your Lab 7 folder. Check this. If you do not have a project folder named array_list, close this project and repeat Step 2.
3. Download file main.c, array_list.c, array_list.h and sput.h from cuLearn. Move these files into your array_list folder.
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.
5. Build the project. It should build without any compilation or linking errors.
6. Execute the project. The test harness will report several errors as it runs, which is what we'd expect, because you haven't started working on the functions the harness tests.

7. Open array_list.c in the editor. Design and code the functions described in Exercises 1 through 8.

Exercise 1

In last week's lab, you learned how to dynamically allocate a structure on the heap; for example:,

```
struct fraction {
      int num;
      int den;
};

typedef struct fraction fraction_t;

fraction_t *pf;
pf = malloc(sizeof(fraction_t)); // pf points to the fraction_t
                                 // structure on the heap
assert(pf != NULL);
```

In a recent lecture, you learned how dynamically allocate an array on the heap; for example, here is the code to allocate an array that has the capacity to hold 100 integers:

```
int *pa;
pa = malloc(100 * sizeof(int));  // pa points to the first
                                 // element in the array, which
                                 // is on the heap
assert(pa != NULL);
```

The data structure that underlies our list collection will combine these two concepts. It will consist of a dynamically-allocated structure, and one of structure's members will be a pointer to a dynamically-allocated array.

Open array_list.h. This file contains the declaration for a structure named `IntList`:

```
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;
```

Notice that the type of member `elems` is "pointer to `int`". This member will be initialized with a pointer to a dynamically-allocated array of integers.

In array_list.c (<u>not</u> array_list.h) define a function that returns a pointer to a new, empty list of integers with a specified capacity. The function prototype is:

```
IntList *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 a dynamically-allocated array with the specified capacity.
- The other block is the dynamically-allocated `IntList` structure. Your `intlist_construct` function will return the pointer to this structure. Remember to save the pointer to the array in member `elems`, and to initialize the `capacity` and `size` members.

The function must terminate (via `assert`) if memory cannot be allocated.

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_construct` function passes all the tests in test suite #1 before you start Exercise 2.

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)
```

(`const` is a reserved word in C. Because parameter `list` has been declared to be `const`, if the function contains code that modifies the `IntList` structure that `list` points to, we'll get a compilation error.)

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: If `list` is a pointer to an `IntList` structure:

```
IntList *list;
```

element `i` in the list can be accessed by this expression:

```
list->elems[i]
```

This expression might appear complicated, so let's break the expression into pieces.

- Parameter `list` is a pointer to the list; i.e., a pointer to an `IntList` structure.
- Recall that the expression `list->elems` is equivalent to `(*list).elems`; that is, we're selecting the `elems` member in the structure pointed to by `list`. Member `elems` is a pointer to an array of integers, so the expression `list->elems` yields the pointer to the array.
- Because `elems` is a pointer to an array, we can access individual elements using the `[]` operator. So, `list->elems[i]` is element `i` in the array that is pointed to by `list->elems`.

Build your project, correcting any compilation errors, then execute the project.

File main.c contains a function that exercises `intlist_print`. The test function does not determine if the information printed by `intlist_print` is correct. Instead, it displays what a correct implementation of `intlist_print` should print (the expected output), followed by the actual output from your implementation of the function. You have to compare the expected and actual output to determine if your function is correct.

Inspect the console output and verify that your `intlist_print` function is correct before you start Exercise 3.

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`.

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 #2 before you start Exercise 4.

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`.

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_capacity` function passes all the tests in test suite # 3 before you start Exercise 5.

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`.

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_size` function passes all the tests in test suite #4 before you start Exercise 6.

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`.

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_get` function passes all the tests in test suite #5 before you start Exercise 7.

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`.

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_set` function passes all the tests in test suite #6 before you start Exercise 8.

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`.

Example, suppose you (1) call `intlist_construct` to allocate a new, empty list; (2) append three integers; and (3) call `intlist_removeall`. When this function returns, the list contains 0 integers.

Hint: this function should not free any of the memory that was allocated by `intlist_construct`, or call `malloc`.

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_removeall` function passes all the tests in test suite #7.

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.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, 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 before you save it. **Do not use any other name for your zip file** (e.g., lab7.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).

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

    3.1. Click the Submit Lab 7 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.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".

Reminder: You'll need your array_list module for Lab 8. This lab assumes your module passes all the tests in the Lab 7 test harness. Remember to complete any unfinished exercises before your next lab period.