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

Lab 7 - Developing a List Collection, First Iteration

Note: This is a long lab, and we expect that many students will not be able to finish it in the two-hour lab period. Take the time you require to do a good job - it's better to write three or four "industrial-strength" functions than rushing to complete all of the exercises by the end of the lab. Any unfinished exercises should be treated as "homework" and should be completed by Lab 8.

Objective

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

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.)

Background

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

- An array's capacity is specified when the array is declared. This capacity is fixed; there is no way to increase an array's capacity at run-time. Also, there no way for a program to "ask" an array for its capacity.
- C does not keep track of how many values are currently stored in an array. It is the programmer's responsibility to do this (often by using an auxiliary variable to count of the number of items in the array). For example, suppose your program declares an array with capacity 10 named `a`:

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

  Your program then 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 for your code to "ask" this array how many array elements have been initialized. In other words, your program has to "remember" that integers have been stored in the first five array elements, meaning that next uninitialized array element is `a[5]`.

- C does not check for out-of-bounds array indices, which means code can access memory outside the array by selecting an out-of-bounds array element; for example, `a[-1]` or `a[10]` will compile without error. At run-time, these expressions do not cause the program to terminate with an error.

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, 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 lists:

- A list can be thought of as a *dynamic array*, which 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. This means that we normally do not need to know the capacity of a list.

- 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 in many programming languages will 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 several 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. Initially, lists will be implemented as dynamically-allocated arrays and will have fixed capacity. We're going to focus on developing functions that provide some common list operations. You'll refine and extend your module in subsequent labs; e.g., you'll improve the design of the underlying data structure, then use this change as the basis for making several improvements to the list operations.

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
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 is 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.

For those students who already know C or C++: when writing the functions, do not use structs. They aren't necessary for this lab.

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:

- intlist.c contains incomplete definitions of several functions you have to design and code.;
- intlist.h contains declaration (function prototypes) for the functions you'll implement. **Do not modify intlist.h.**
- main.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.**

Instructions

1. Launch Pelles C and create a new Pelles C project named intlist. The project type must be Win32 Console program (EXE). You should now have a folder named intlist.
2. Download file main.c, intlist.c and intlist.h from cuLearn. Move these files into your intlist folder. **You must also add main.c and intlist.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 intlist.c. Pelles C will automatically add intlist.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. The harness will report several errors while it tests the functions in intlist.c, which is what we'd expect,

because you haven't started working on these functions.

5. Open main.c. Read the comments and code for function `intlist_max`. This function is passed an array of integers and returns the largest element in the array. Notice that first parameter is declared as `int *list`, instead of the equivalent `int list[]`. Notice how this function uses a "walk-the-pointer" algorithm to traverse the array, and uses pointer dereferencing to access array elements, instead of using the `[]` operator. Make sure you understand this approach - you'll need it in several of the exercises.

Exercise 1
In intlist.c, define a function that returns a pointer to a dynamically allocated array of integers with the specified capacity. The function prototype is:

```
int *intlist_create(int capacity);
```

This function terminates (via `assert`) if the capacity is less than or equal to 0 or if the required amount of memory can't be allocated.

Hint: if you're stuck, there is a similar function in the lecture slides, which was presented in class.

Exercise 2
In intlist.c, define a function that prints the integers stored in list (array) pointed to by parameter `list_ptr`. The function prototype is:

```
void intlist_print(int *list_ptr, int len);
```

Parameter `len` is the length (size) of the list; i.e., the number of integers stored in the array. This function should assume that 0 <= `len` <= the list's capacity.

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, and the second argument (parameter `len`) is 4, 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: `[]`.

This specification continues on the next page.

Your function must use a "walking pointer" algorithm. You are **not** permitted to use the `[]` operator to access the array elements; expressions of the form `list_ptr[i]` are not allowed.

Hint: you wrote a similar function (`print_array`) in Lab 4.

Exercise 3
In intlist.c, define a function that returns the index (position) of the first occurrence of an integer equal to `target` in the list (array) pointed to by `list_ptr`. The function prototype is:

```
int intlist_index(int *list_ptr, int len, int target);
```

Parameter `len` is the length (size) of the list; i.e., the number of integers stored in the array. This function should assume that 0 <= `len` <= the list's capacity.

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

If `target` is not in the list, the function should return -1.

Your function must use a "walking pointer" algorithm. You are **not** permitted to use the `[]` operator to access the array elements; expressions of the form `list_ptr[i]` are not allowed.

Exercise 4
In intlist.c, define a function that counts the number of integers in the list (array) pointed to by `list_ptr` that are equal to `target`, and returns that number. The function prototype is:

```
int intlist_count(int *list_ptr, int len, int target);
```

Parameter `len` is the length (size) of the list; i.e., the number of integers stored in the array. This function should assume that 0 <= `len` <= the list's capacity.

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

Your function must use a "walking pointer" algorithm. You are **not** permitted to use the `[]` operator to access the array elements; expressions of the form `list_ptr[i]` are not allowed.

Exercise 5
In intlist.c, define a function that determines if an integer in the list (array) pointed to by `list_ptr` is equal to `target`. The function prototype is:

```
_Bool intlist_contains(int *list_ptr, int len, int target);
```

Parameter `len` is the length (size) of the list; i.e., the number of integers stored in the array. This function should assume that 0 <= `len` <= the list's capacity.

This specification continues on the next page.

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_ptr` 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 6
In intlist.c, define a function that appends an integer to the end of list (array) pointed to by `list_ptr`. The function prototype is:

```
int intlist_append(int *list_ptr, int len, int capacity, int val);
```

Parameter `len` is the length (size) of the list; i.e., the number of integers stored in the array. This function should assume that 0 <= `len` <= the list's capacity.

Parameter `capacity` is the list's capacity.

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

If `val` was appended, the function should return the length of the modified list. If the function was not successful, because the list was full, it should leave the list unchanged and return -1. This return value indicates to the caller that it is an error to attempt to append a value to a full list.

You are permitted to use the `[]` operator to access the array elements; expressions of the form `list_ptr[i]` are allowed.

Note: the test harness will check if the integer returned by `intlist_append` (the length of the modified list, or -1 if the list was not changed) is correct. The test code will also display the list before and after `intlist_append` is called. You will have to look at this output and verify manually if the modified list is correct.

Exercise 7
In intlist.c, define a function that deletes the integer at the specified position in the list (array) pointed to by `list_ptr`. The function prototype is:

```
int intlist_delete(int *list_ptr, int len, int index);
```

Parameter `len` is the length (size) of the list; i.e., the number of integers stored in the array. This function should assume that 0 <= `len` <= the list's capacity.

Parameter `index` is the index (position) of the integer that should be removed. If a list contains `len` integers, valid indices range from 0 to `len` - 1.

This specification continues on the next page.

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

If the integer was deleted, this function should return the length of the modified list. 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 -1.

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.

You are permitted to use the `[]` operator to access the array elements; expressions of the form `list_ptr[i]` are allowed.

Note: the test harness will check if the integer returned by `intlist_delete` (the length of the modified list, or -1 if the list was not changed) is correct. The test code will also display the list before and after `intlist_delete` is called. You will have to look at this output and verify manually if the modified list is correct.

Exercise 8
In intlist.c, define a function that removes the first occurrence of an integer equal to `target` in list pointed to by `list_ptr`. The function prototype is:

```
int intlist_remove(int *list_ptr, int len, int target);
```

Parameter `len` is the length (size) of the list; i.e., the number of integers stored in the array. This function should assume that 0 <= `len` <= the list's capacity.

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

If `target` is found and removed from the list, this function should return the length (size) of the modified list. 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 -1.
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].

Note: the test harness will check if the integer returned by `intlist_remove` (the length of the modified list, or -1 if the list was not changed) is correct. The test code will also display the list before and after `intlist_remove` is called. You will have to look at this output and verify manually if the modified list is correct.

This specification continues on the next page.

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

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 intlist.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 intlist).

3. Log in to cuLearn, click the Submit Lab 7 link and submit intlist.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".

Challenge Exercise

Modify your solutions to Exercises 6 (`intlist_append`) and 7 (`intlist_delete`) to use pointers to access the array elements; that it, don't use expressions of the form `list_ptr[i]` to retrieve or modify array elements.

Hint: if variable `p` points to the first element of an array; i.e., contains the address of element 0, then the expression `(p + i)` points to (is the address of) element *i*. If variable `p1` points to the first element of an array and variable `p2` points to element *i*, then the expression `(p2 - p1)` yields the value *i* (the index of the element pointed to by `p2`).

Edited: Monday, October 21, 2013