

Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Winter 2017

Lab 7 - Developing a List Collection, First Iteration

Objective

In this lab, you'll begin the development of a C module that implements a list collection that is similar to a Python list or a Java ArrayList. You'll learn how we can build useful data structures in C by combining `structs`, pointers to `structs`, dynamically allocated arrays and pointers to arrays.

Attendance/Demo

To receive credit for this lab, you must demonstrate your solutions to the exercises. When you have finished all the exercises, call a TA, who will review your code, ask you to run the test harness provided on cuLearn, and assign a grade. For those who don't finish early, a TA will ask you to demonstrate the functions 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; for example, this statement declares an array that holds 10 integers: `int numbers[10];`

This capacity is fixed; there is no way to increase the array's capacity at run-time.

- C doesn't provide an operator or standard library function that returns an array's capacity. The idiom:

```
int capacity = sizeof(arr) / sizeof(arr[i]);
```

doesn't work when `arr` is a function array parameter; for example, this function is incorrect:

```
/* Initialize all elements of arr to 0. */
void initialize(int arr[])
{
    int capacity = sizeof(arr) / sizeof(arr[i]); //No!
    for (int i = 0; i < capacity; i = i + 1) {
        arr[i] = 0;
    }
}
```

This technique doesn't work because parameter `arr` is not an array; it's a pointer to an

array, so `sizeof(arr)` yields the size of the pointer, not the total size of the array pointed by `arr`.

- C doesn't check for out-of-bounds array indices, which means code can access memory outside the array by using an out-of-bounds index. If `numbers` is declared this way:

```
int numbers[10];
```

the expressions `numbers[-1]` or `numbers[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 the array.

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 C 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.
- 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.
- List operations 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 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.

The data structure that underlies this collection is known as a *dynamic array* or *growable array*. This is an array that increases its capacity when it becomes full. C doesn't provide dynamic arrays, so you're going to learn how to build them "from scratch", using C's fixed capacity arrays.

In the first version of the list 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 the core list operations. You'll refine and extend your module in a subsequent lab.

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

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.**

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

1. Launch Pelles C and create a new Pelles C project named `array_list`.
 - 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`.

2. Download file `main.c`, `array_list.c`, `array_list.h` and `sput.h` from cuLearn. Move these files into your `array_list` folder.
3. 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`.

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`.

4. Build the project. It should build without any compilation or linking errors.
5. 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.
6. Open `array_list.c` in the editor and do Exercises 1 through 8.

Exercise 1

In a recent lecture, you learned how dynamically allocate an array on the heap; for example, this code fragment allocates an array that has the capacity to hold 100 integers and initializes all of the array elements to 0:

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

assert(pa != NULL);
for (int i = 0, i < 100; i += 1) {
    pa[i] = 0;
}
```

In another lecture, you learned how to dynamically allocate a structure on the heap; for example, here is the code to allocate a structure that stores the Cartesian coordinates of a point:

```

typedef struct {
    int x;
    int y;
} point_t;

point_t *pt;
pt = malloc(sizeof(point_t)); // pt points to the point_t
                               // structure on the heap

assert(pt != 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 type named `intlist_t`:

```

typedef struct {
    int *elems; // Pointer to backing array.
    int capacity; // Maximum number of elements in the list.
    int size; // Current number of elements in the list.
} intlist_t;

```

Notice that the type of member `elems` is "pointer to `int`". This member will be initialized with a pointer to an array of integers that has been allocated from the heap.

In `array_list.c` (not `array_list.h`) you have been provided with an incomplete definition of a function named `intlist_construct` that, when completed, will return a pointer to a new, empty list of integers with a specified capacity. The function prototype is:

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

You must modify the function so that it correctly implements all of the following requirements:

- The function terminates (via `assert`) if `capacity` is less than or equal to 0.
- The function allocates two blocks of memory from the heap:
 - One block is the list's backing array; that is, a dynamically-allocated array with the specified capacity.
 - The other block is the dynamically-allocated `intlist_t` structure. Your `intlist_construct` function will return the pointer to this structure. Remember to save the pointer to the backing array in member `elems`, and to initialize the `capacity` and `size` members.
- The function terminates (via `assert`) if memory cannot be allocated for the structure or the array.

Complete the function definition. If you're not sure how to do this, reread the lecture slides on dynamic memory allocation and the two examples presented at the start of this exercise.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Review the console output, and verify that your `intlist_construct` function passes all the tests in test suite #1 before you start Exercise 2.

Interlude (read this before attempting the remaining exercises)

All the functions in Exercises 2 through 8 have a parameter of type `intlist_t *`:

```
return_type fn_name(intlist_t *list, ...);
```

In other words, the first argument passed to these functions is a pointer to a list (an `intlist_t` structure).

The function can access element `i` in the list's backing array by using this expression:

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

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

- Parameter `list` is a pointer to the list; i.e., a pointer to an `intlist_t` 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 `int`; specifically, it points to the first element in an array of integers. Therefore, the expression `list->elems` yields the pointer to the first element in the list's backing array.
- Because `elems` is a pointer to an array, we can access individual elements using the `[]` operator. So, `list->elems[i]` is element at position `i` in the array that is pointed to by `list->elems`; in other words, element `i` in the list's backing array.

Exercise 2

In `array_list.c`, you have been provided with an incomplete definition of a function that prints the integers stored in a list. The function prototype is:

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

(Recall that `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_t` structure that `list` points to, we'll get a compilation error.)

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

Complete the function definition. The required format for the output is:

`[elem0 elem1 elem2 ... elemn-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: `[]`.

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

Test suite #2 exercises `intlist_print`, but it cannot verify if the information printed by the function 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.

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

Exercise 3

In `array_list.c`, you have been provided with an incomplete definition of a function that appends an integer to the end of a list. The function prototype is:

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

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

Parameter `element` contains the value that will be appended to the list. 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`.

Complete the function definition.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Review the console output, and verify that your `intlist_append` function passes all the tests in test suite #3 before you start Exercise 4.

Exercise 4

In `array_list.c`, you have been provided with an incomplete definition of a function that returns the capacity of a list. The function prototype is:

`int intlist_capacity(const intlist_t *list)`

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

Complete the function definition.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Review the console output, and verify that your `intlist_capacity` function passes all the tests in test suite # 4 before you start Exercise 5.

Exercise 5

In `array_list.c`, you have been provided with an incomplete definition of a function that returns the size of a specified list. The function prototype is:

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

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

Complete the function definition.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Review the console output, and verify that your `intlist_size` function passes all the tests in test suite #5 before you start Exercise 6.

Exercise 6

In `array_list.c`, you have been provided with an incomplete definition of a function that returns the element located at a specified index (position) in a list. The function prototype is:

```
int intlist_get(const intlist_t *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`, inclusive.

Complete the function definition.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Review the console output, and verify that your `intlist_get` function passes all the tests in test suite #6 before you start Exercise 7.

Exercise 7

In `array_list.c`, you have been provided with an incomplete definition of a function that stores an integer at a specified index (position) in a list. The function will return the integer that was previously stored at that index. The function prototype is:

```
int intlist_set(intlist_t *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`, inclusive.

Parameter `element` contains the value that will be stored in the list.

Complete the function definition.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Review the console output, and verify that your `intlist_set` function passes all the tests in test suite #7 before you start Exercise 8.

Exercise 8

In `array_list.c`, you have been provided with an incomplete definition of a function that empties a list. The function prototype is:

```
void intlist_removeall(intlist_t *list)
```

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

Suppose you (1) call `intlist_construct` to allocate a new, empty list; (2) append three integers; and (3) call `intlist_removeall`, for example:

```
intlist_t *my_list = intlist_construct(10); // capacity 10, size 0
_Bool success;

success = intlist_append(my_list, 2); // capacity 10, size 1
success = intlist_append(my_list, 4); // capacity 10, size 2
success = intlist_append(my_list, 6); // capacity 10, size 3
intlist_removeall(my_list); // capacity 10, size 0
```

When `intlist_removeall` returns, the list contains 0 integers.

Complete the function definition. 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. Review the console output, and verify that your `intlist_removeall` function passes all the tests in test suite #8.

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 grading/sign-out 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.
3. **You'll need your `array_list` module for Lab 8. That 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.**

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 `structs` and arrays, using the same notation as C Tutor. This exercise is intended to help you visualize what happens when programs allocate memory from the heap.

1. The *Labs* section on cuLearn has a link, [Open C Tutor in a new window](#). Click on this link.
2. Copy the `intlist_t` declaration from `array_list.h` and your solutions to Exercises 1 and 3-8 to C Tutor.
3. Write a short `main` function that exercises your list functions.
4. *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.
5. 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 visualization displayed by C Tutor.