Lab 8 - Developing a List Collection

Objective

In this lab, you will develop a C module that lets programs create lists of integers. You will learn how we can build useful data structures in C by using pointers to connect dynamically allocated arrays and structs.

Background

C arrays have several limitations:

- An array's capacity is specified at compile-time; for example,

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

  declares an array named `numbers` that holds 10 integers. The array's capacity cannot be changed at run-time.

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

- 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. Expressions such as `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*. Python has a built-in type (class) named `list` and Java provides a class named `ArrayList`. 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 list collections 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 generate a run-time error if you specify an invalid list index. By default, this normally results in an error message being displayed, then the program terminates.

- In Python, several common operations on lists are provided by built-in operators, functions and methods, which means that programmers don't have to implement these operations "from scratch". Java's `ArrayList` class defines several methods that provide similar operations. Compare this with C arrays, which provide very few built-in array operations.

In this lab, you are going to develop a C module that implements a "subset" of Python's built-in `list` type. This will be a useful module to have in your "toolbox" if you end up doing a lot of C programming.

We will not attempt to implement all the features of Python's `list` type. Although a C list will be based on a dynamically-allocated array, it will have fixed capacity; in other words, it won't grow when it becomes full. We are going to focus on developing functions that provide the core list operations, but several list operations won't be supported.

We will 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

None of the functions you write should call `calloc`, `realloc` or `free`. Only `list_construct` (Exercise 1) is permitted to call `malloc`.

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). Instructions were given in the VS Code installation instructions.

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. Do not leave testing until after you have written all your functions.

Getting Started

Step 1: Depending on your operating system, download (**lab8_win.zip** or **lab8_mac.zip**) the starting code from Brightspace and extract the zipped file.

Step 2: Double click on lab8.code-workspace

Step 3: lab8_xxx folder contains the following files:

- array_list.c contains incomplete definitions of several functions you have to design and code;
- array_list.h contains the declaration of the `list_t` struct, as well as declarations (function prototypes) for the functions you will 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.**

Step 4: Run the project. It should build without any compilation or linking errors.

Step 5: Execute the project. For the functions where the test harness is provided (the functions in main.c) will report several errors as it runs, which is what we'd expect, because you haven't started working on the functions.

Step 6: Open array_list.c in the editor and do Exercises 1 through 10. Do not make any changes to main.c, array_list.h or sput.h. All the code you will write must be in array_list.c.

Exercise 1

In a recent lecture, you learned how to dynamically allocate memory on the heap. For example, this code fragment allocates an array that has the capacity to hold 100 integers and stores the pointer to the array in variable `pa`. It then initializes all the array elements to 0:

```
int *pa;
pa = malloc(100 * sizeof(int));  // pa points to the first
                                 // element in the array
assert(pa != NULL);
for (int i = 0, i < 100; i += 1) {
    pa[i] = 0;
}
```

This code fragment allocates a `struct` 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
                              // struct in 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 `struct`, and one of the struct's members will be a pointer to a dynamically-allocated array.

Open array_list.h. This file contains the declaration for a `struct` named `list_t`:

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

Notice that the type of member `elems` is "pointer to `int`". This member will be initialized with the 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 `list_construct` that, when completed, will return a pointer to a new, empty list of integers with a specified capacity. The function prototype is:

```
list_t *list_construct(int capacity);
```

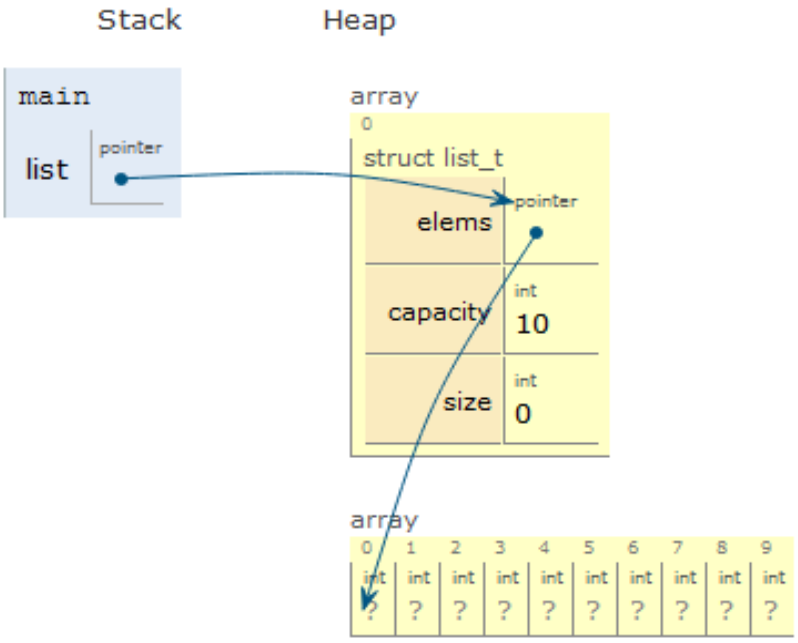
You must modify the function so that it correctly implements all of the following requirements:

- 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 `list_t struct`. Your `list_construct` function will return the pointer to this `struct`.
- The function terminates (via `assert`) if memory cannot be allocated for the `struct` or the array.
- The function initializes the `struct`'s `elems`, `capacity` and `size` members with the pointer to the backing array, the list's capacity and the list's size, respect

Suppose `main` contains this statement:

```
list_t *list = list_construct(10);
```

Here's a C Tutor diagram that depicts memory after this statement is executed:



Notice how the `struct`'s `elems`, `capacity` and `size` members are initialized.

Complete the function definition. Use the console output produced by the test harness to help you identify and correct any flaws. Verify that `list_construct` passes all the tests in test suite #1. If `list_construct` fails any tests, we recommend that you use C Tutor to help you debug your code. Copy the definition of `list_construct` to the C Tutor editor. You'll also need to copy/paste the declaration of the `list_t` struct (from array_list.h) to C Tutor and write a short main function that calls `list_construct`. Execute the code, step-by-step. When `list_construct` is correct, the image displayed by C Tutor should be similar to the screen capture shown above.

Interlude (read this before attempting the remaining exercises)

Function `list_print` has been defined for you. This function is passed a pointer to a list, and prints the list using the format:

> [$elem_0$ $elem_1$ ... $elem_{n-1}$], capacity: *capacity*, size, *size*

For example, if `list_print` is passed a list with capacity 10 that contains 1, 5, -3 and 9, the function will display:

```
[1 5 -3 9], capacity: 10, size: 4
```

Feel free to call `list_print` to help you debug your code.

All the functions in Exercises 2 through 10 have a parameter of type `list_t *`:

> *return_type fn_name*(list_t *list, ...);

In other words, the first argument passed to these functions is a pointer to a list.

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 a `list_t struct`.
- Recall that the expression `list->elems` is equivalent to `(*list).elems`; that is, we're selecting the `elems` member in the `struct` 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 the 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

File array_list.c contains an incomplete definition of a function named `list_append` that appends an integer to the end of a list. The function prototype is:

```
_Bool list_append(list_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 the list is not full. (In other words, `element` will be stored in the list only if it has room for at least one more element.) 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. Use the console output to help you identify and correct any flaws. Verify that `list_append` passes all the tests in test suite #2 before you start Exercise 3.

Exercise 3

File array_list.c contains an incomplete definition of a function named `list_capacity` that returns the capacity of a list. The function prototype is:

```
int list_capacity(const list_t *list)
```

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

Complete the function definition. Use the console output to help you identify and correct any flaws. Verify that `list_capacity` passes all the tests in test suite #3 before you start Exercise 4.

Exercise 4

File array_list.c contains an incomplete definition of a function named `list_size` that returns the size of a specified list. The function prototype is:

```
int list_size(const list_t *list)
```

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

Complete the function definition. Use the console output to help you identify and correct any flaws. Verify that `list_size` passes all the tests in test suite #4 before you start Exercise 5.

Exercise 5

File array_list.c contains an incomplete definition of a function named `list_get` that returns the element located at a specified index (position) in a list. The function prototype is:

```
int list_get(const list_t *list, int index)
```

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

Complete the function definition. Use the console output to help you identify and correct any flaws. Verify that `list_get` passes all the tests in test suite #5 before you start Exercise 5.

Exercise 6

File array_list.c contains an incomplete definition of a function named `list_set` 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 list_set(list_t *list, int index, int element)
```

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

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

Complete the function definition. Use the console output to help you identify and correct any flaws. Verify that `list_set` passes all the tests in test suite #6 before you start Exercise 7.

Exercise 7

File array_list.c contains an incomplete definition of a function named `list_removeall` that empties a list. The function prototype is:

```
void list_removeall(list_t *list)
```

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

To illustrate the use of this function, here is a code fragment that allocates a new, empty list. It appends three integers to the list, then calls `list_removeall`:

```
list_t *my_list = list_construct(10); // list has capacity 10, size 0
_Bool success;

success = list_append(my_list, 2); // list has capacity 10, size 1
success = list_append(my_list, 4); // list has capacity 10, size 2
success = list_append(my_list, 6); // list has capacity 10, size 3
list_removeall(my_list);           // list has capacity 10, size 0
```

When `list_removeall` returns, the list has room for 10 integers.

Complete the function definition. This function should not free any of the memory that was allocated by `list_construct`, or call `malloc`.

Use the console output to help you identify and correct any flaws. Verify that `list_removeall` passes all the tests in test suite #7 before you start Exercise 8.

Exercise 8

File array_list.c contains an incomplete definition of a function named `list_index`. This function returns the index (position) of the first occurrence of a specified "target" integer in a list. If the integer is not in the list, the function should return -1.

The function prototype is:

```
int list_index(const list_t *list, int target)
```

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

Complete the function definition. Use the console output to help you identify and correct any flaws. Verify that `list_index` passes all the tests in test suite #8 before you start Exercise 9.

Exercise 9

File array_list.c contains an incomplete definition of a function named `list_count`. This function counts the number of times that a specified "target" integer occurs in a list, and returns that number. The function prototype is:

```
int list_count(const list_t *list, int target);
```

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

Complete the function definition. Use the console output to help you identify and correct any flaws. Verify that `list_count` passes all the tests in test suite #9 before you start Exercise 10.

Exercise 10

File array_list.c contains an incomplete definition of a function named `list_contains`. This function returns `true` if a list contains a specified "target" integer; otherwise it should return `false`. The function prototype is:

```
_Bool list_contains(const list_t *list, int target);
```

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

Complete the function definition. Use the console output to help you identify and correct any flaws. Verify that `list_contains` passes all the tests in test suite #10.

Wrap-up

Submit array_list.c to Brightspace.

Before submitting your lab work:

- Review *Important Considerations When Submitting Files to Brightspace* (lab3 file). Remember: submit early, submit often.
- Make sure that fraction.c has been formatted to use K&R style or BSD/Allman style, as explained in *General Requirements*.
- Ensure you are submitting the file that contains your solutions, and not the unmodified file you downloaded from Brightspace!

You are permitted to make changes to your solutions and resubmit the files as many times as you want, up to the deadline. You need to demo your work to TAs during the lab for grades.

Homework Exercise - Visualizing Program Execution

In the final exam, you will be expected to understand and draw diagrams that depict the execution of short C programs that use dynamically allocated memory, using the same notation as C Tutor. This exercise is intended to help you develop your code tracing/visualization skills.

1. Click on this link to open C Tutor.
2. Copy/paste the `list_t` declaration from array_list.h and your solutions to Exercises 1 through 10 into the C Tutor editor. You can comment out the `assert` calls.
3. Write a short `main` function that exercises your functions.
4. *Without using C Tutor*, trace the execution of your program. Draw memory diagrams that depict the program's activation frames just before each of your functions returns. Use the same notation as C Tutor.
5. Use C Tutor to trace your program one statement at a time, stopping just before each function returns. Compare your diagrams to the visualizations displayed by C Tutor. If C Tutor complains that your program is too long, delete the comments above the function definitions.
6. Do not submit your solutions for this exercise.