Project 3: List

Objectives

- Use structs and pointers
- Practice with pointers
- Use a procedural language in an Object Oriented way
- Several styles of testing software in C
- Practice dynamically allocating memory **and** freeing it
- Use industry standard tools to prevent memory errors.

Introduction

In this project we will complete a data structure using C. The data structure we will complete is a List represented using an array, mostly modeled after the **ArrayList** class found in the Java Standard Library. In doing so, you will gain experience with pointers and managing memory. We utilize several different methods of testing to ensure your list operates in a robust and correct manner. The project is completed in three parts:

1. Creating an object to store in the List
2. Programming the List
3. Testing the List

Starter Files

- Makefile
- rubric.txt
- String.h
- String.c
- List.h
- List.c
- string-test.c
- simple-test.c
- list-test.c
- random-test.c

Tasks

Part 1: Struct practice: A String object

- **`String.h` contains the specification** for a Java-like representation of a `String` in C. There is a struct definition, which holds a pointer to a character array, and the length of the array.

- **Start implementing `String.c`** by writing the functions that Create and Destroy the String object. `CreateString` is analogous to a constructor function in Java. It must allocate space for the `String` struct, then allocate space for a copy of the C-string that it takes as parameter.

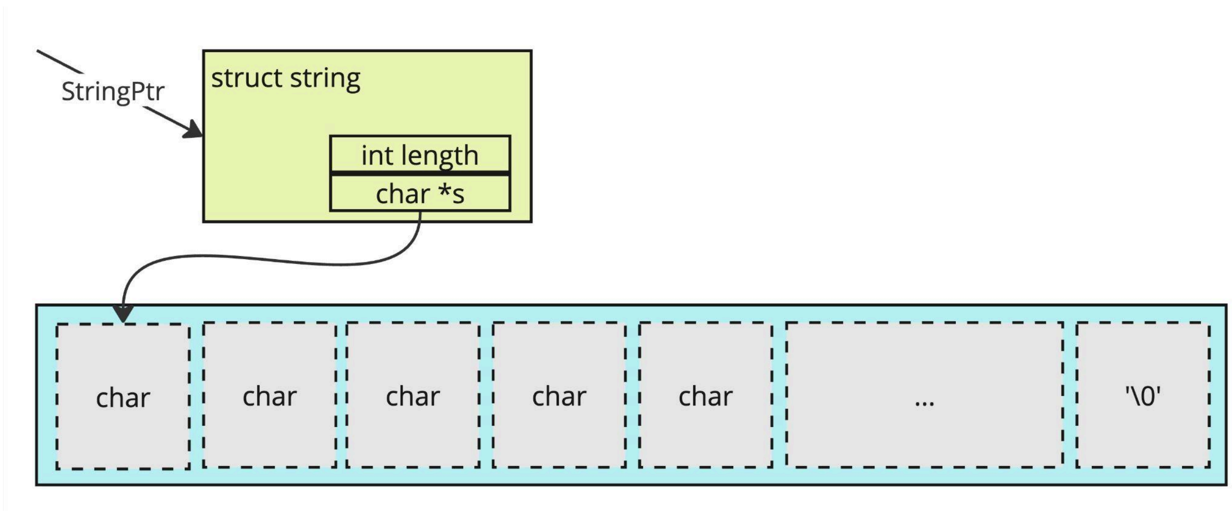
- **Complete the other three methods**: `CharAt`, `StringCompare`, and `StringToString`. These will be very similar to the C-string manipulation functions we covered in Module 2. You can now use the length property of the `String` struct to help prevent exceeding the buffer length.

- **You may not use the functions found in <string.h>**. If you need helper functions like `strcmp` or `strcpy`, write your own versions. *Style note*: “private” helper functions should be declared in String.c with the `static` keyword.

- **String-test.c contains some basic unit tests**. When they pass, String.c is good enough for you to move to Part 2. Reminder, run the unit tests with Valgrind!

```
$ valgrind ./string-test
```

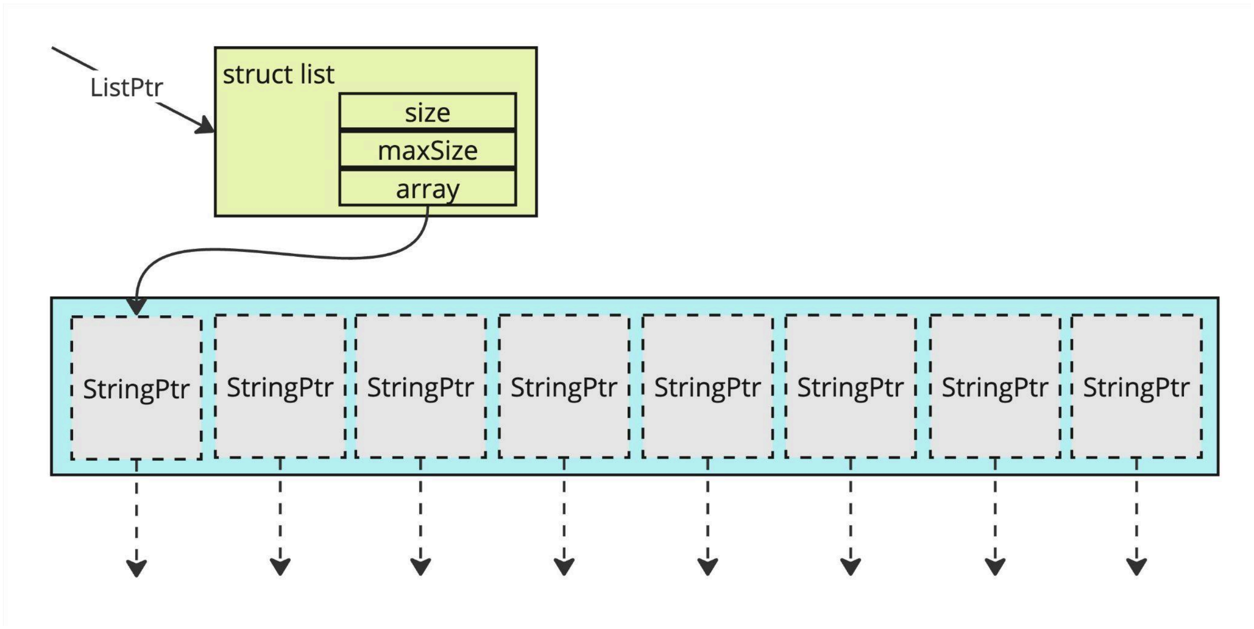
- Use this graphical depiction of our String object to help visualize what is going on:



Part 2: The List Data Structure

Complete the List given the specification in `List.h`.

1. Before you start modifying the code, read through all the files and make sure that you understand how they work. The files you need to change are:
    a. List.c
    b. list-test.c
2. Study the graphical diagram of the List.



3. Implement the CreateList function and the corresponding DestroyList function in List.c. To do so, dynamically allocate an List struct using malloc:

    ```
    ListPtr L = (ListPtr) malloc(sizeof(ListPtr));
    ```

    Our List will be a list of Strings from String.h. Therefore, an array of StringPtrs will be the underlying data structure in the List. This array is declared:

    ```
    StringPtr *array
    ```

    This of course means the real type is `String **array`. A Pointer to an array of String Pointers! Allocate this array dynamically using `malloc`. When Destroying the list, remember that each call to `malloc()` should have an accompanying call to `free()`.

4. **List Basic Operations**. `List.h` defines a variety of methods for using our data structure. Among them are methods that simply return values to the caller from the List struct or its array.

```
int ListSize(ListPtr L);
bool ListIsEmpty(ListPtr L);
StringPtr ListGet(ListPtr L, int index);
int ListIndexOf(ListPtr L, String element);
```

Implementing these is more straightforward than methods that modify the List, so this is a good place to start.

Recall: you should use the -> operator to dereference a value from a struct pointer.

5. **List Mutation Operations**. The specification also includes methods for changing the List by adding and removing elements. Work on these next.

```
void ListClear(ListPtr L);
void ListAdd(ListPtr L, int index, StringPtr element);
void ListAppend(ListPtr L, StringPtr element);
StringPtr ListRemove(ListPtr L, int index);
```

6. **Sorting the List.** The specification includes a method called ListSort.

   To implement this, you should sort the Strings contained in the List using `StringCompare`, the function you wrote in `String.c`.

   This isn't an algorithms class, so only translate the code for *insertion sort* from Java to C.

   To use `StringCompare`, substitute `(A[j] > key)` with a call to `StringCompare` in the `while` loop condition.

```
public void insertion_sort(int A[], int n) {
   for (int i = 1; i < n; i++) {
```

```
            int key = A[i];
            int j = i - 1;
            while ((j >= 0) && (A[j] > key)) {
                A[j + 1] = A[j];
                j--;
            }
            A[j + 1] = key;
        }
    }
```

If you would like, you can use a faster sorting algorithm in place of Insertion Sort for 1 point extra credit.

7. **Expanding the Array**. Our List should **double** the underlying array in size each time it runs out of space. This keeps the List working regardless of how many Strings it contains. The List does not ever need to shrink, but you could consider restoring its original size if it is cleared.
   Hint: check the `malloc` man page for a function called `realloc`. Otherwise, you will need to free any old memory.

8. **Printing the List.** Use the `StringToString` function in `String.c` to implement the `PrintList` function. Don't forget that `StringToString` allocates memory that needs to be freed when you are done. The `PrintList` function simply prints the contents of the list, one per line.

List Extra Credit [13 points]

The methods that modify the List (`ListAdd`, `ListAppend`, `ListRemove`) can be implemented in constant time (that is, O(1) worst case time). Do so for `ListAdd` and `ListRemove` for 1 point of extra credit each.

Additionally, use a different sorting algorithm for `ListSort` for another 1 point of extra credit. I don't care which algorithm you use.

Because we are sorting Strings, the values can be sorted with a potentially more efficient algorithm, **radix sort**. Implement radix sort for 5 points extra credit!

I asked ChatGPT for an implementation of Radix Sort, you can find that here.

I will award you an additional 5 points extra credit to modify this to use dynamically allocated intermediate arrays! This is more efficient.

DO NOT ATTEMPT RADIX SORT UNTIL EVERYTHING ELSE IS COMPLETE, INCLUDING PART 3

Part 3: Testing the List

Your project will be evaluated on the basis of it passing your own tests, as well as mine.

Smoke Testing

Smoke testing is the metaphor of turning a machine on to see if it catches fire. `simple-test.c` contains a basic smoke test of the functionality of `List.c`. A segmentation fault is like it catching fire (to use the metaphor). Use this for the most simple tests. Don't use the other test programs until this one works.

```
$ make simple-test
$./simple-test 10
~~ output here ~~
```

Unit Testing

`list-test.c` already has a few unit tests. This test file is incomplete as it does not test all possible boundary conditions. Your job is to expand it to ensure that your list operates as expected. This could take a few tests. I think about 10-15 is good, and more is okay too.

Unit tests each test one function at a time independent of the rest of the functions. You may have to build some helper functions to be able to execute each method independently.

Random Testing

We have provided a completed test program `random-test.c`, which does random testing of the various List operations. This simulates how an application might use the List with a large number of operations. The random test program could also be used to assess the performance of your list implementation.

```
Usage: random-test <list size> [<test size=list size>] [<seed>]
```

The `random-test.c` program shown above takes up to three arguments. The first argument is required. The second argument is the number of random operations to perform. It is optional. The default value if not supplied by the user is the same as the initial size of the list. The third argument is the random number generator seed. By setting a seed, you will get the same random number sequence, which can be helpful with debugging. Otherwise the random number generator sets a different seed for each run of the program.

Memory Tests

When you have the unit tests complete, and `random-test` runs without failure, its time pass valgrind.

```
$ valgrind --leak-check=full --track-origins=yes ./random-test 10000 10000 123
```

Should be sufficient to prove the List works.

You can execute this from the Makefile with:

```
$ make mem-test-list
```

Differences from the zybook 9.8 vector Example

Object oriented programming in C has been around for a long time.
There are many ways to do it!

Here are some key differences in these projects from the Vector program shown in Chapter 9.8 of Zybooks.

- Our List will store pointers to other structs, not integers
- The names of each method in our List are modeled after the Java ArrayList specification.
- The typedefs used in section 9.8 are different from the ones provided in our project.