Lab 3

Pointer Manipulation Lab

Due: Week of February 13, before the start of your lab section

This is an individual-effort project. You may discuss concepts and syntax with other students, but you may discuss solutions only with the professor and the TAs. Sharing code with or copying code from another student or the internet is prohibited.

The purpose of this assignment is to give you more confidence in C programming and to begin your exposure to pointers and to file input/output.

The instructions are written assuming you will edit and run the code on your account on the *csce.unl.edu* Linux server. If you wish, you may edit and run the code in a different environment; be sure that your compiler suppresses no warnings, and that if you are using an IDE that it is configured for C and not C++.

Learning Objectives

After successful completion of this assignment, students will be able to:

- Recognize the hazards of indeterminate values.
- Use C's string functions from `string.h`[1].
- Use C's file I/O functions from `stdio.h`[2].
- Alias and reassign pointers.
- Create and traverse a linked list.

Continuing Forward

Being able to understand the mistakes in Sections 2.1 and 2.2 will help you avoid them in future labs. Being able to work with pointers – that is, with variables that hold addresses – will help you specifically in future labs that use pointers but more generally in future labs that require you to think about accessing memory.

[1] See §7.8.1 and §B.3 of Kernighan & Ritchie's *The C Programming Language*, 2nd ed.
[2] See §7.5, §7.7, and §B1.1 *The C Programming Language*, 2nd ed.

During Lab Time

During your lab period, the TAs will provide a refresher on linked lists and will describe Insertion Sort. The TAs will also describe some string functions and some I/O functions from C's standard library. During the remaining time, the TAs will be available to answer questions.

Before leaving lab, *at a minimum* complete Sections 2–3.2.4.

No Spaghetti Code Allowed

In the interest of keeping your code readable, you may *not* use any **`goto`** statements, nor may you use any **`continue`** statements, nor may you use any **`break`** statements to exit from a loop, nor may you have any functions **`return`** from within a loop.

Scenario

Working at the Pleistocene Petting Zoo certainly is proving to be interesting. You're glad that you don't have to worry about the problem of the giant sloths very slowly chasing their handlers, but now it seems that Archie has decided to try to write a program or two. At a glance, his code is smellier than the wooly rhinoceros' enclosure. But you take a closer look anyway to try to understand why his code acts strangely.

1 Assignment Summary

This assignment is principally about getting comfortable when explicitly working with memory. Being able to think about a value and a reference to that value distinctly will improve your programming skills in any language.

Before you do so, in Section 2 you will examine Archie's code. Parts of Archie's programs use code that the C standard explicitly states will result in undefined behavior. By understanding the mistakes that Archie made, we hope that you can avoid them in your own code.

In Section 3, you will build and use a linked list. This will require you to allocate space for the list's nodes and manipulate pointers that connect the nodes to each other.

There are no particular restrictions in this assignment other than those common to most lab assignments in this course. You can check whether you're using a **`goto`** or **`continue`** statement, or whether you're using **`break`** or **`return`** to exit a loop, by running the constraint-checking Python script: `python constraint-check.py linkedlistlab.json`

2 Stray Values in Memory

2.1 Pleistocene Petting Zoo Marquee

Archie shows you his first program, which he hoped would be used to greet guests, but it doesn't always work right:

```
1  /*****************************************************************
2   * This program will output
3   **          Welcome to the
4   **     Pleistocene Petting Zoo!
5   **
6   ** Get ready for hands-on excitement on the count of three! 1.. 2.. 3..
7   ** Have fun!
8   * With brief pauses during the "Get ready" line.
9   *****************************************************************/
10
11 #include <stdio.h>
12 #include <unistd.h>
13
14 void splash_screen() {
15     const char *first_line = "\t     Welcome to the\n";
16     const char *second_line = "\tPleistocene Petting Zoo!\n";
17     printf("%s%s\n", first_line, second_line);
18 }
19
20 void count() {
21     int i;
22     sleep(1);
23     printf("Get ready for hands-on excitement on the count of three! ");
24     while (i < 3) {
25         fflush(stdout);
26         sleep(1);
27         i++;
28         printf("%d.. ", i);
29     }
30     printf("\nHave fun!\n");
31 }
32
33 int main() {
34     splash_screen();
35     count();
36     return 0;
37 }
```

Sometimes the output was what he expected:

```
      Welcome to the
  Pleistocene Petting Zoo!

Get ready for hands-on excitement on the count of three! 1.. 2.. 3..
Have fun!
```

But sometimes the output was missing the "`1..  2..  3..`":

```
      Welcome to the
  Pleistocene Petting Zoo!

Get ready for hands-on excitement on the count of three!
Have fun!
```

What mistake did Archie make? What change to *one* line will fix Archie's bug? Place your answers in *answers.txt*.

2.2 Math Doesn't Work Right . . . Or Does It?

"Wow," Archie says. "I can't believe that I missed that. Maybe you can help me with some other code that I wrote for one of my other start-ups. Here it is." Archie shows you the code:

```
1  /*****************************************************************
2   * This program will add two numbers and then it will multiply two other
3   * numbers. Finally, it will subtract the second result from the first
4   * result.
5   *****************************************************************/
6
7  #include <stdio.h>
8
9  int *add(int a, int b) {
10     int result = a + b;
11     return &result;
12 }
13
14 int *multiply(int p, int q) {
15     int result = p * q;
16     return &result;
17 }
18
19 int main() {
20     int *sum = add(4, 5);
```

```
21      printf("sum = %d\n", *sum);
22      int *product = multiply(2, 3);
23      printf("product = %d\n", *product);
24      printf("sum - product = %d - %d = %d\n",
25                              *sum, *product, *sum - *product);
26      return 0;
27  }
```

Archie explains that when he compiles the program with the **clang** compiler and then runs it, he gets this output:

```
sum = 9
product = 6
sum - product = 6 - 6 = 0
```

And when he compiles the program with the **gcc** compiler and then runs it, the program terminates with a segmentation fault.

"I see that one compiler is giving me an incorrect answer, and the other compiler is telling me that I'm using memory in an unsafe way – but what am I doing wrong, and why does it produce an incorrect answer?"

What mistake did Archie make? Why does `*sum` have the value 6 on line 25? Why does `*sum - *product` produce the value 0? Place your answer in *answers.txt*.

3 Challenge and Response

You plug in your shiny, new keyboard, tune your satellite radio to the Greatest Hits of the 1920s, and settle in to solving a more interesting problem.

To protect against corporate espionage, you are responsible for writing code for a challenge-and-response system. Anybody can challenge anyone else in the Pleistocene Petting Zoo's non-public areas by providing the name of a book and a word contained within the book, and the person being challenged must respond with another word from that book, based on certain rules:

- All of the book's words are sorted alphabetically without regard to capitalization (for example, "hello" occurs after "Hear" and before "HELP")
- The challenge word occurs *occurrences* times in the book
- If *occurrences* is an odd number then the response word is the word *occurrences* places **after** the challenge word in the alphabetized list; if the challenge word is less than *occurrences* places from the end of the list then "wrap around" to the start of the list and resume counting

- If *occurrences* is an even number then the response word is the word is $2 \times occurrences - 1$ places **after** the challenge word in the alphabetized list; if the challenge word is less than $2 \times occurrences - 1$ places from the end of the list then "wrap around" to the start of the list and resume counting

Here is a simple example. Suppose the words in the specified book are:

word	*occurrences*
apple	7
banana	4
carrot	15
date	3
eggplant	2
fig	6
granola	9
horseradish	9
ice	6
jelly	3
kale	1
lemon	2
mango	8
naan	7
orange	5
pineapple	1
quinoa	11
raisin	4
spaghetti	10
tomato	12

If the challenge word is "horseradish" then because horseradish occurs 9 times in the book, the response word is "quinoa," which is 9 places in the list after "horseradish." If the challenge word is "eggplant" then the response is "horseradish," which is 3 places ($2 \times 2 - 1 = 3$) later in the list than "eggplant." If the challenge word is "quinoa" then the response word is "horseradish," because the response word should be 11 places after "quinoa," but the end of the list is 3 places later; "horseradish" is in position 8 in the list ($11 - 3 = 8$). (*Note:* "horseradish" and "quinoa" being each other's response words is coincidental. Unusual things happen with short lists of words that do not generalize to longer lists.)

You break the problem down into four sub-problems:

1. Designing the Data Structure and Its Algorithms
2. Alphabetizing Words
3. Inserting Words
4. Responding to a Challenge

The Books

Four small "books" are included with the starter code:

- "Animals" (sorted, 7 words)
- "Plants" (unsorted, 7 words)
- "Cars" (sorted, 74 words)
- "Food" (unsorted, 125 words)

Two real books have also been reduced to one word per line:[3]

- Mary Shelly's *Frankenstein; Or, The Modern Prometheus* (filename "Frankenstein") https://www.gutenberg.org/ebooks/84 (sorted, 74,363 words)
- Arthur Conan Doyle's *The Lost World* (filename "TheLostWorld") https://www.gutenberg.org/ebooks/139 (unsorted, 77,268 words)

The very small files of 7 words can be useful for debugging, and the moderate-sized files of 74–125 words should give you confidence in the correctness of your solution. The real books of more than 74,000 words will be useful to reveal whether you have any memory leaks in your code. The files marked as *sorted* have all of their words already in alphabetically sorted order, ignoring capitalization; the files marked as *unsorted* do not have their words sorted (the words in "Plants" and "Food" are in a randomly-selected order; the words in "TheLostWorld" appear in the order that they appear in the original *The Lost World*).

Each book file, "*file*", has a corresponding "*file*-table.md" that contains a Markdown-formatted table of the challenge words, the number of occurrences for each challenge word, and the corresponding response word. You may use these files to confirm the correctness of your solution.

Throughout the assignment, we note that if building the list takes more than a few seconds, there is a bug in your code; for context, we can build the list for *Frankenstein* in 1–2 seconds and the list for *The Lost World* in 1.5–3 seconds. We can locate a word (or determine the absence of a word) in the *Frankenstein* list in under 0.5ms and in the *The Lost World* list in under 0.9ms. Your code may take longer, but it should not take much longer.

You will earn most of the credit for this lab if your code works for pre-sorted files of up to 200 words. The remaining credit is for making your code work with unsorted files and, when using files of up to 80,000 words, your code can generate a list and find a word in fewer than 20 seconds.

[3]The text for these books was obtained from Project Gutenberg. In accordance with Paragraph 1.C of the Project Gutenberg License, all references to Project Gutenberg have been removed from the "derived works" that we are distributing. (Removing the references to Project Gutenberg was also necessary to ensure that *only* the words from the books are used for the challenge-and-response system.)

3.1 Differences and Similarities between Java and C that are Relevant to this Assignment

In some regards, Java keeps things simple: every variable is a reference, except when it isn't. In other regards, C keeps things simple: you always know whether the variable you're using is a value or a pointer.

3.1.1 Comparing Strings

You probably learned that when comparing Java Strings, using the equality operator `==` is error-prone. When comparing two String literals (or variables assigned to String literals), the equality operator usually acts as a naive programmer would expect: `"abc" == "abc"` evaluates to **`true`**. When one or both of the Strings are generated at runtime, such as from user input, then the equality operator rarely evaluates to **`true`**: `userInput == "abc"` will evaluate to **`false`** even when the user entered "abc".

The reason for this is that when comparing objects (other than boxed types), Java's comparators compare the objects' references; that is, Java comparators compare the objects' memory addresses. Using the `==` operator to compare Strings evaluates to **`true`** only when the two Strings occupy the same address; that is, they are both literally the same String object. This is why you were taught to use Java's **`String.equals()`** method to compare strings.

Comparing C strings' variables has the same pitfall: because the string variables are pointers to the first character in their respective strings, using arithmetic comparators will compare the strings' addresses. If you want to compare two C strings, you would use the **`strcmp()`**[4] function. The wrinkle is that **`strcmp()`** returns `0` (*i.e.*, **`false`**) when the two strings are equal; you will often see the idiom **`if`** `(!strcmp(string1, string2)) {`.

The **`strcmp(string1, string2)`** function actually performs a lexicographic comparison of the strings, returning a negative value if `string1` occurs alphabetically earlier than `string2`, zero if every character in the two strings match, and a positive value if `string1` occurs alphabetically later than `string2`. In this regard, C's **`strcmp()`** function is more like Java's **`String.compareTo()`** method than **`String.equals()`**.

3.1.2 Copying Strings

Because Java Strings are immutable objects, you can safely copy a string by simply copying its reference (this is called *aliasing*). You can safely write the statement `string1 = string2;` without worrying about changes to `string2` causing changes in `string1` (if you were to make changes to `string2`, it would result in a new String object being assigned to the `string2` variable).

For mutable objects, creating an alias (that is, copying the reference) results in the situation that changes made through one variable are visible through the other variable. For example, if you have the statements `list1 = list2; list2.add(foo);` then

[4] See footnote 1.

`list1.size() == list2.size() && list1.contains(foo)` will evaluate to **`true`**. If the object's class implements the `Cloneable` interface then you can make a copy of an object without aliasing it. If you have the statements `list1 = list2.clone(); list2.add(foo);` then `list1.size() == list2.size()` will evaluate to **`false`**.

In general, C strings are mutable.[5] This means that you generally don't want to create an alias.[6] Instead, use the **`strcpy(destination, source)`** or **`strncpy(destination, source, n)`**[7] function to copy the `source` string into the memory pointed to by `destination`. The **`strcpy()`** function will continue copying until encountering the terminating `NUL` in the `source` string – this is very slightly faster (not enough that you'd notice) but is safe only if you can prove that `destination` has enough memory allocated for the string. The **`strncpy()`** function will copy until encountering the terminating `NUL` or until it has copied $n-1$ characters (after which it will append a terminating `NUL`) – this is safer because you can ensure that the string copied to `destination` will fit within the space allocated for it.

3.1.3 Allocating and Deallocating Memory

As we'll note in Section 3.2.3, Java's **`new`** keyword allocates space for the new object, inferring the amount of space needed based on the class's definition. In C, you use the **`malloc()`** function to allocate space[8], and you must be explicit about how much space you need. An idiom is to combine **`malloc()`** with the **`sizeof()`** function, as you saw in PokerLab, and as you'll see near the start of Section 3.2.4.

Java uses a *garbage collector* to reclaim memory allocated for objects that are no longer in use. The unpredictability of when garbage collection happens makes an automatic garbage collector unsuitable for many of C's uses. For this reason (among others), the programmer is responsible for deallocating memory that is no longer needed. This is done with the **`free()`** function.

While a variable will go out of scope at the end of the code block in which it was declared, memory allocated in that code block persists unless explicitly **`free`**d. Once the last pointer pointing to that memory goes out of scope, you no longer have a way to **`free`** that memory, resulting in a *memory leak*. On the other hand, **`free`**ing memory while it is still being used by another pointer can result in undefined behavior. This requires careful thought to make sure that you **`free`** all memory that you allocated, but only after it is safe to do so.

For many short-running programs, such as those you often write in school, you often can ignore the need to **`free`** allocated memory since all the program's memory will be reclaimed by the operating system when the program terminates. A member of the C Standard Committee recently described this as having a maid that will clean up your mess.[9]

[5] The exceptions are string literals, which are immutable, and strings declared as a pointer to a constant, which if treated as mutable will result in undefined behavior.

[6] Sometimes you can't create an alias. If the left-hand-side of an assignment is a constant pointer or is effectively a constant pointer – such as an array inside a struct – then it cannot be re-assigned.

[7] See footnote 1.

[8] There are a small handful of alternate functions, each with their own use cases, but **`malloc()`** is most-suitable for this lab.

[9] https://twitter.com/__phantomderp/status/1619322783162568705,

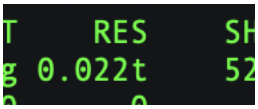

Figure 1: Screenshot showing a student's program consuming 0.022 terabytes of physical memory; the allocated virtual memory is not visible in the image.

I advise you not to rely on that "maid" even for a "short-running program," such as this one. In an earlier version of this lab, there were a dozen or so students whose code, when tested against a 75,000-word file, would quickly consume all the server's physical memory. As the first of these programs thrashed the virtual memory system, it prevented other services from working effectively, including the one that I had precautionarily introduced to kill a test after a couple of minutes. It consumed enough resources that the system administrator couldn't log in to determine why the server had slowed to a crawl. As I was already logged in, I was able to kill the process as the system administrator was preparing to disconnect the server from the power line. The system administrator later commented about the resources it consumed, "You ought never to see a 'T' in the memory column" (Figure 1).

3.1.4 Idioms for Defining and Initializing `Struct` Types

In PokerLab, you worked with a struct that was defined by **`typedef`** as a new type. In this lab, you will work with a struct defined simply as a **`struct`**, without a **`typedef`**. There are compelling arguments for and against both approaches in different use cases. You should learn to be comfortable with both approaches.

In this lab, the function that initializes a node is also responsible for allocating space for that node. In PokerLab, you saw the other idiom, in which the calling function is responsible for allocating space and passing the pointer to the initializer (this pointer is analogous to Java's implicit **`this`** parameter). The approach of having the caller allocate space seems to be more common, but both approaches are common enough to be aware of both. (The third idiom is not to have a separate initializing function; I discourage this approach.)

3.2 Designing the Data Structure and Its Algorithms

You decide that a circular linked list is the best data structure option for the challenge-and-response system. You probably learned about linked lists in CSCE 156, RAIK 184H, or SOFT 161; however, we will provide a refresher.

3.2.1 Singly-Linked List

A *linked list* is a linear collection of data. Like an array, each element (or *node*) has a particular position in the list, and when you iterate over the list, you always access the

https://twitter.com/__phantomderp/status/1619323139665846273

elements in the same order every time (unless you change or re-order the elements).

In an array, the elements are contiguous in memory, and you can access a specific element by indexing the array (or, equivalently, performing pointer arithmetic). In a linked list, however, the nodes can be in arbitrary locations in memory, and the nodes are connected by references (in C, pointers). You can access a specific element only by following pointers from one node to the next until you reach the desired node.

The simplest linked list is a *singly-linked list*. A node consists of a *payload* (the data that we care about) and a reference to the *next* node; see Figure 2.

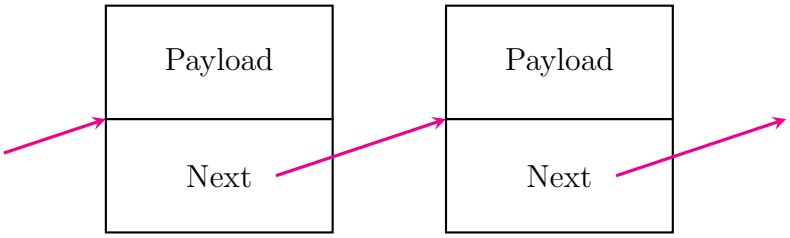

Figure 2: Nodes in a singly-linked list consist of the payload data and a reference that points to the next node.

A linked list's greatest advantage over an array is that inserting and removing a node at an arbitrary location takes constant time, whereas inserting an element into an array (assuming there is sufficient memory allocated for the array) or removing an element from an array requires moving all the elements that follow the element's index. Inserting a new node, C between adjacent nodes A and B (where $B = A.next$) requires connecting $C.next$ to B and re-assigning $A.next$ to C; see Figure 3.

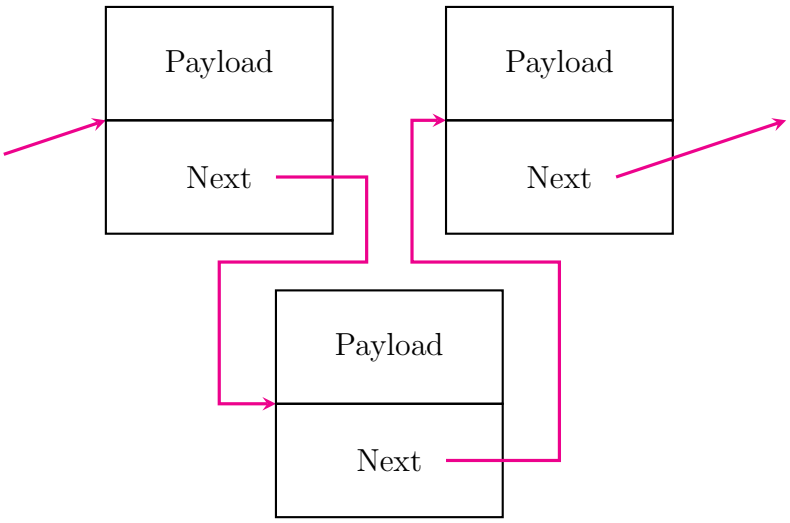

Figure 3: Inserting a new node into a singly-linked list only requires assignments to the affected *next* pointers.

As with an array, you do need to maintain a variable that points to the list. Conventionally, this is a reference to the *head* of the list. (Note that if a new node is inserted before the current head node, then the new node becomes the head of the list, and your `head` variable would need to be updated.) It is not uncommon to also maintain a reference to the *tail* of the list.

3.2.2 Circular Linked List

A *circular linked list* is a linked list in which the tail's *next* field points to the head of the list. In essence, a circular linked list has no head (because every node is some node's *next*), and has no tail (because every node's *next* is non-NULL); see Figure 4.

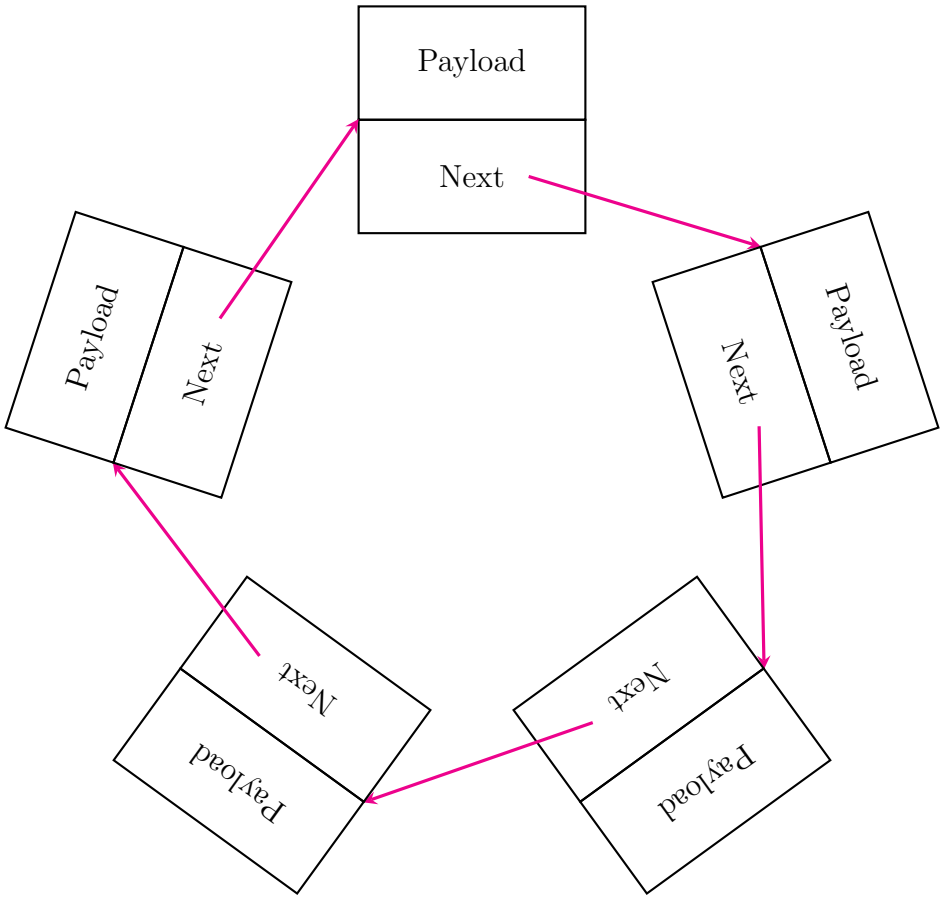

Figure 4: A circular linked list does not have a well-defined head and tail.

You still need to maintain a variable that points to *some* node in the list.

3.2.3 Equivalent Java Code

In Java, you probably wouldn't implement your own linked list; instead, you would use `java.util.LinkedList`, which has been available since J2SE 1.2 (ignoring for the moment that Java's standard library doesn't have a circular linked list). A list of hypothetical `Payload` objects would be created with:

```
List<Payload> payloads = new LinkedList<>;
```

C doesn't have a built-in linked list data type, so you will need to design one. Let us consider what a custom linked list would look like in Java.

```
 1 public class Node {
 2     private final String word;
 3     private int occurrences;
 4     private Node next;
 5     private Node previous;
 6
 7     public Node(String word) {
...         ...
12     }
13
14     public void insertAfter(Node existingNode) {
...         ...
23     }
24     ...
99 }
```

Creating and inserting a new node would look something like this:

```
200     Node node = new Node("eggplant");
201     Node otherNode = ... // code to determine where the new node goes
202     node.insertAfter(otherNode);
```

Recall that in Java, all variables except primitive types (such as `occurrences` on line 3) are references. This means that the `next` field on line 4 is a reference to another Node, just as we described in Section 3.2.1. The payload is the `word` and how many `occurrences` the word has, exactly what we need for the challenge-and-response system.

Recall also that in Java, the **`new`** keyword allocates space for the new object, and the constructor call – `Node("eggplant")` – initializes the object.

3.2.4 C Implementation

In *challenge-response.h*, you'll see a **`struct`** with the same fields as our Java example:

```
36 struct node {
37     char word[MAXIMUM_WORD_LENGTH];
38     int occurrences;
```

```
39      struct node *next;
40      struct node *previous;
41  };
```

In *challenge-response.c*, you'll also see the **create_node()** function:

```
23
24  /* Allocates memory for a new node to hold the word, and initializes the
25   * node's fields. Returns a pointer to the new node. */
26  struct node *create_node(const char *word) {
27      struct node *new_node = malloc(sizeof(struct node));
28      /* WRITE THE REST OF THIS FUNCTION */
29      return new_node;
```

As you can see, it allocates space for a new node using **malloc()**. The code that you will need to add to it will copy the `word` argument into the `word` field and set an appropriate initial value for the `occurrences` field. Since we don't yet know where this node will go, set the node's `next` pointer to point to the node itself.

The other function you need to write now is **insert_after()**:

```
31
32  /* Inserts new_node into the list after existing_node; that is, new_node
33   * becomes existing_node's "next". If existing_node's original "next" is
34   * non-NULL, then that will become new_node's "next". */
35  void insert_after(struct node *existing_node, struct node *new_node) {
36      /* WRITE THIS FUNCTION */
```

As the name and documentation indicate, you need to add code that will update the nodes' `next` pointers so that `new_node` is placed in the list immediately after `existing_node`. You can ignore the `previous` pointers for this assignment.

Two notes:

- The header comment notes that "If `existing_node`'s original `next` is non-NULL, then that will become `new_node`'s `next`." In the case of the circular linked list used in this lab, `existing_node`'s `next` pointer *must* be non-NULL, and so you do not need to check for NULL `next` pointers.
- There is another function, **insert_before()**, which you do not need to implement in this lab.

After you have implemented **create_node()** and **insert_after()**, go to the **main()** function in *linkedlistlab.c* and un-comment the call to **test_linked_list_functions()**.

```
58  int main() {
59  //      test_linked_list_functions();
```

Build the executable with the command: `make`. Be sure to fix both errors and warnings. When the program compiles without generating any warnings or errors, run it with the

command `./linkedlistlab`. The output should indicate a list with the nodes in the order of "first node," "fourth node," "second node," and "third node."

3.3 Change All Uppercase Letters to Lowercase

Add code to **`word_to_lowercase()`** that calls that function to convert all letters in a word to lowercase letters. Unlike KeyboardLab, you may use C's **`tolower()`** function[10] to convert uppercase letters to lowercase letters.

```
48
49 /* Returns a copy of the word that has all uppercase letters replaced
50  * with lowercase letters. The original string is unchanged. */
51 char *word_to_lowercase(const char *word) {
52     /* WRITE THIS FUNCTION */
53     return NULL;
54 }
55
56 /* Compares two words based on alphabetical order.
57  * Returns a negative value if word1 occurs alphabetically before word2.
58  * Returns a positive value if word1 occurs alphabetically after word2.
59  * Returns 0 if the two words are identical.
60  * This function is really just a wrapper around strncmp. */
61 int compare_words(const char *word1, const char *word2) {
62     return strncmp(word1, word2, MAXIMUM_WORD_LENGTH);
```

The starter code includes a function to compare two words (you do not need to write this function) but it assumes that both words are completely lowercase.

3.4 Inserting Words

Comment-out (or delete) the call to **`test_linked_list_functions()`**.

For this sub-problem, the user will be prompted to enter the name of a book, which will be the filename of a file that contains all of the book's words. All punctuation has already been removed from the files, and each line in the file contains exactly one word. For this assignment, you only need to work with files whose contents are already sorted.

```
67
68 /* Determines if the word is already in the list. If it is, then the
69  * number of occurrences for that word is incremented. If it isn't, then
70  * a new node is created for the word and inserted into the list at the
71  * correct alphabetic location. Returns a pointer to the head of the
72  * list, which is either the original head or a node containing the word
73  * (if the word occurs before the original head's word or if the
```

[10] See §7.8.2 and §B.2 of Kernighan & Ritchie's *The C Programming Language*, 2nd ed.

```
74  * original head is NULL).
75  * NOTE: THIS FUNCTION ASSUMES THAT THE HEAD OF THE LIST IS THE NODE WITH
76  * THE ALPHABETICALLY-EARLIEST WORD! */
77 struct node *insert_word(struct node *head, const char *word) {
78     /* WRITE THIS FUNCTION */
79     return NULL;
80 }
81
82 /* Given the name of the book file from the user, reads the file
83  * line-by-line. Under the assumption that there is exactly one word per
84  * line and that all punctuation has been removed, this function builds
85  * a doubly-linked list of the words in alphabetical order, keeping
86  * track (as part of a node's payload) how many times each word occurs
87  * in the file. */
88 struct node *build_list(const char *filename) {
89     /* WRITE THE REST OF THIS FUNCTION */
90     return NULL;
```

Add code to **insert_word()**[11] and **build_list()** to read the specified file one line at a time.[12] For each word, convert it to lowercase, and then traverse the list to find the appropriate place for the word. (Note that there will not be a list to traverse when your code reads the first word!) If the word is not in the list then create a node for that word and insert it into the list at the correct location. If there is already a node containing that word, then increment that node's variable that tracks the number of occurrences. Be sure that *only* the word is placed in a node; specifically, do not include a newline character nor any other characters that are not part of the word.

Note that when you are working with files whose contents are already sorted, you are guaranteed that every word read from the file (other than the first word) is either another instance of the previous word that was read, or it will appear in the list immediately after the previous word that was read. You might use this characteristic to simplify the code to build your list for now. Later, in Section 3.6.2, you will modify your code to work with unsorted files. For now, however, you can earn most of the assignment's points by working with the simplifying assumption of having pre-sorted files.

Build the program and correct all warnings and errors. When the program compiles without generating any warnings or errors, run it. If your program requires more than a few seconds to build the list, there is a bug in your code. If your program does not produce the expected output, the **print_list()** utility function will help you see the list that your code created.

[11] **insert_word()**'s header comment says that the head of the list is the node with alphabetically-earliest word. For a circular linked list, you can write a perfectly-functional implementation with the head at anyplace in the list, but our grading software assumes that the head is the alphabetically-earliest word. If you write code that meets the specification but is not scored correctly, see the syllabus for grade challenges.

[12] See §7.5 and §B1.1.1 of Kernighan & Ritchie's *The C Programming Language*, 2nd ed. for **fopen()** and **fclose()**, and §7.7 and §B1.1.4 for **fgets()**.

3.5 Respond to a Challenge

You now have implemented enough of the other sub-problems that you can write the code to respond to a challenge.

```
95
96  /* Given an alphabetically sorted list of words with the number of
97   * occurrences of each word, and given the challenge_word, will return
98   * the response word based on the following rules:
99   * - If the number of occurrences is an even number then the response
100  *   word is 2*occurrences-1 *after* challenge_word in the list
101  * - If the number of occurrences is an odd number then the response
102  *   word is that many places *after* challenge_word in the list
103  * - If the challenge_word is fewer than that number of places from
104  *   the end of the list, then "wrap around" to the start of the list
105  *   and resume counting
106  * - If challenge_word is not present in the list, then the response is
107  *   "<challenge_word> is not present!" */
108 char *respond(const struct node *list, const char *challenge_word) {
109     /* WRITE THIS FUNCTION */
110     return NULL;
111 }
```

After the word list is complete (after you have inserted all words in the file), the user will be prompted to enter the challenge word. Add code to **`respond()`** that traverses the word list to find that word. If the word is not present in the list, return "(word) is not present!", where "(word)" is the challenge word.

Use the number of occurrences recorded in that the challenge word's node to find the response word as described in the challenge-and-response rules, and return that word.

If your program does not provide the response word nearly instantaneously, there is a bug in your code.

3.6 The Final Touches

You have now completed enough of the assignment to receive at least 19 out of 25 points. The remaining steps will re-visit your earlier work to make improvements. Because you will be modifying your working code, this would be a good time to make a backup copy of your code or to commit it to a private Git repository.

3.6.1 Designing the Data Structure and Its Algorithms, Revisited: Doubly-Linked List

You may be able to produce small performance improvements if you can traverse the list forward and backward by creating a circular doubly-linked list instead of a circular singly-linked list.

A *doubly-linked list* is a linked list with the property that each node maintains a link not only to the `next` node but also a link to the `previous` node. In C, these links are pointers.

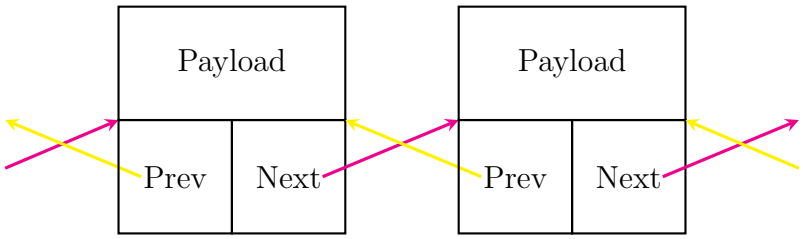

Figure 5: Nodes in a doubly-linked list consist of the payload data and references that point to the previous and next nodes.

Inserting new node, C between adjacent nodes A and B (where $B = A.next$ and $A = B.previous$) requires connecting $C.previous$ to A and $C.next$ to B, and re-assigning $A.next$ to C and $B.previous$ to C.

Modify your **`insert_after()`** function to update not only the `next` pointers but also the `previous` pointers so that `new_node` is placed between `existing_node` and the node that originally was located immediately after `existing_node`. Modify your **`insert_after()`** and **`respond()`** functions to take advantage of the `previous` pointers.

After the program compiles without warnings or errors, you may want to use **`print_list`** to confirm that the `previous` pointers are updated correctly. If your program requires more than a few seconds to build the list, there is a bug in your code.

Creating a circular doubly-linked list is worth another point. This would be a good time to make a backup copy of your code or to commit it to a private Git repository.

3.6.2 Inserting Words, Revisited: Working with Unsorted Files

In Section 3.4, you had the option of simplifying your code by treating the last word added as the head of the list instead of traversing the list to find the correct location for a word. To work with unsorted files, you will need a way to build and maintain a sorted list – the insertion sort algorithm is a good choice. You might also want to modify your code to create a circular doubly-linked list.

Insertion Sort While you probably learned about sorting in CSCE 156, RAIK 184H, or SOFT 161, you may not have learned about *Insertion Sort*. If you did learn about Insertion Sort, you probably learned that it's a $\mathcal{O}(n^2)$ algorithm that is less efficient than $\mathcal{O}(n \log n)$ sorting algorithms such as Merge Sort and Quick Sort. Insertion Sort has a particular advantage in that it can be applied *as the list is built*, making for a much simpler and less error-prone implementation than a different sort that requires the list to already be built.

The Insertion Sort algorithm reads an input and then traverses a sorted list to find the proper location in the sorted list for the input. The input is then inserted into the list at that location.

Your current implementation of **`insert_word()`** might be based off of the assumption that the word is either another instance of the last word to be added to the list or is not present in the list. Update **`insert_word()`** so that it looks for the word in the sorted list. If the word is found in its proper location in the sorted list, then update the number of occurrences as before. If the word is not present in its proper location in the sorted list, then insert a new node for that word at its proper location in the sorted list. If your program requires more than a few seconds to build the list, there is a bug in your code.

Building a list of up to 200 words and generating the correct response when the "book" file is unsorted is worth two points (regardless of whether the list is singly- or doubly-linked). This would be a good time to make a backup copy of your code or to commit it to a private Git repository.

3.6.3 Inserting Words Revisited: Working with Large Files

Your code might already be able to handle large files in just a few seconds, in which case you are finished.

On the other hand, your code might run briskly when working with smaller files of only a couple of hundred words but then become very sluggish when the number of words is in the thousands. This tends to be due to one (or both) of two causes. Look for inefficient algorithms, and look for memory leaks. If you use the `top` utility while running your program, and you notice that your program is allocating more than 10MB, then you probably have a memory leak. If your program is allocating more than 1GB, then your program most certainly has a memory leak.

Being able to generate a list from a file of up to 80,000 pre-sorted words and generating the correct response, all within 20 seconds, is worth two points. Being able to do so from a file of up to 80,000 unsorted words is worth another point.

4 Turn-in and Grading

When you have completed this assignment, upload *challenge-response.c* to Canvas.

No Credit for Uncompilable Code

If the TA cannot create an executable from your code, then your code will be assumed to have no functionality.[13] Before turning in your code, be sure to compile and test your code on your account on the *csce.unl.edu* Linux server with the original driver code, the original header file(s), and the original *Makefile*.

[13] At the TA's discretion, if they can make your code compile with *one* edit (such as introducing a missing semicolon) then they may do so and then assess a 10% penalty on the resulting score. The TA is under no obligation to do so, and you should not rely on the TA's willingness to edit your code for grading. If there are multiple options for a single edit that would make your code compile, there is no guarantee that the TA will select the option that would maximize your score.

Late Submissions

This assignment is due before the start of your lab section. The due date in Canvas is five minutes after that, which is ample time for you to arrive to lab and then discover that you'd forgotten to turn in your work without Canvas reporting your work as having been turned in late. We will accept late turn-ins up to one hour late, assessing a 10% penalty on these late submissions. Any work turned in more than one hour late will not be graded.

Rubric

This assignment is worth 25 points.

_____ **+2** Student's answers in *answers.txt* demonstrate an understanding of the bug in Section 2.1's code and how to correct it.

_____ **+3** Student's answer in *answers.txt* demonstrate an understanding of the bug in Section 2.2's code.

_____ **+1** **`create_node`** creates and initializes a **`struct node`** as specified.

`insert_after`:

_____ **+3** correctly places a new node in a list by updating the **`next`** pointers.

_____ **+1** also updates the **`previous`** pointers.

_____ **+2** **`word_to_lowercase`** returns a copy of the input string with uppercase letters replaced with lowercase letters.

`insert_word`:

_____ **+2** creates a new node at the appropriate location in the list when the word is not already present in the list (where the appropriate location is immediately after the last word to have been added).

_____ **+1** does not create a new node but instead updates the number of occurrences, when the word is present in the list.

_____ **+3** **`build_list`** opens a file for reading, builds a list by reading one line at a time and the word that is passed to **`insert_word()`**, and closes the file after the last line has been read, when the words in the file are pre-sorted.

_____ **+2** **`respond()`** produces the correct response word in accordance with the specified rules when the words in the file are pre-sorted.

_____ **+2** Your challenge-response code can build a list and generate the correct response when the words in the file are unsorted.

_____ **+2** Your challenge-response code can generate a list from a file of up to 80,000 words and generate the correct response within 20 seconds when the words in the file are pre-sorted.

_____ **+1** Your challenge-response code can generate a list from a file of up to 80,000 words and generate the correct response within 20 seconds when the words in the file are unsorted.

Penalties

_____ **-1** Newline characters are included in the word strings when building a list.

_____ **-1** for each **`goto`** statement, **`continue`** statement, **`break`** statement used to exit from a loop, or **`return`** statement that occurs within a loop.

Epilogue

You hear somebody enter the room. "*Frankenstein*, 'boat'," is the challenge, and she answers, "borne." Archie introduces you to the new arrival, "Lil, this is our new developer, the one who wrote the app we just used." He turns to you: "This is Lilith Redd from business operations." He turns back to her and continues, "Lil, what's the good word?"

"The word isn't good, I'm afraid. I just heard back from the insurance company."

To be continued...