ECS 34: Programming Assignment #3

Instructor: Aaron Kaloti

Fall 2020

Contents

1 Changelog **1**

2 General Submission Details **1**

3 Grading Breakdown **2**

4 Submitting on Gradescope **2**
4.1 Regarding Autograder . . . 2
4.1.1 Memory Leaks . . . 2
4.1.2 Number of Visible vs. Hidden Cases . . . 2
4.1.3 Test Cases' Inputs . . . 2

5 Conceptual Prerequisites **2**
5.1 `static` Keyword . . . 2
5.2 File I/O . . . 3
5.3 Structs . . . 6

6 Programming Problems **7**
6.1 Restrictions . . . 7
6.2 Part #1 . . . 7
6.3 Part #2 . . . 8
6.4 Part #3 . . . 9
6.5 Part #4 . . . 11

1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.
- v.2: Fixed typo in `test_strtok_c.c` and fixed corresponding output. The issue is shown on line #116 in the example output for part #3; `tok4` is now correctly shown to be `ra` but was originally incorrectly shown to be `a`, because in `test_strtok_c.c`, I had accidentally printed the string referenced by `tok3` instead of the string referenced by `tok4` at one point; I have fixed the C file on Canvas.
- v.3: Added autograder details. Can assume at least one current and at least one previous course in part #4.
- v.4: Added brief mention of autograder visible test case files.

2 General Submission Details

Partnering on this assignment is prohibited. If you have not already, you should read the section on academic misconduct in the syllabus.

This assignment is due the night of Monday, November 9. Gradescope will say 12:30 AM on Tuesday, November 10, due to the "grace period" (as described in the syllabus). *Be careful about relying on the grace period for extra time; this could be risky.*

*This content is protected and may not be shared, uploaded, or distributed.

You should use the `-Wall` and `-Werror` flags when compiling. The autograder will use these flags when it compiles your program.

3 Grading Breakdown

The autograder score will be out of 70 points. Below is the number of points corresponding to each part:

- Part #1: 8
- Part #2: 8
- Part #3: 27
- Part #4: 27
  - `loadStudent()` and `printStudent()`: 15.
  - `freeStudent()`: 6.
  - `areClassmates()`: 6.

4 Submitting on Gradescope

You should only submit your C files with the names indicated below. You may be penalized for submitting additional files. You have infinite submissions until the deadline.

During the 10/02 lecture, I talked about how to change the active submission, just in case that is something that you find yourself wanting to do.

4.1 Regarding Autograder

Your output must match mine *exactly*.

There is a description about how to interpret some of the autograder error messages in the directions for the previous two programming assignments. I will not repeat that description here.

Only submit `prog3.c`.

The autograder takes longer to run than it did in the previous assignments.

4.1.1 Memory Leaks

Some of the test cases automatically fail if your program has a memory leak. These cases will tell you that there is a leak, if this is the case. Note that failing to close a file also causes a memory leak, perhaps because the `FILE` struct has dynamically allocated memory. Memory leaks will be checked with `valgrind`.

4.1.2 Number of Visible vs. Hidden Cases

Below is the breakdown of visible vs. hidden test cases on each part.

Part	No. Cases	No. Visible	No. Hidden
1	4	2	2
2	4	1	3
3	15	3	12
4	8	3	5

4.1.3 Test Cases' Inputs

See `part1_cases.c`, `part2_cases.c`, `part3_cases.c`, and `part4_cases.c` on Canvas. These programs can be compiled similarly to how the example C files below are compiled. Each of these files takes a command-line argument specifying the test case number.

5 Conceptual Prerequisites

5.1 `static` Keyword

The `static` keyword, when used on a local variable, causes the variable to behave like a global variable while still having the scope in which it was declared. A common use of this is to allow a local variable to preserve its value even when the function in which the variable was defined returns. Below is an example.

```
$ cat static1.c
#include <stdio.h>

void foo(void)
{
    int x = 0;
    static int y = 0;
    printf("x=%d, y=%d\n", x, y);
    x += 1;
    y += 1;
}

int main()
{
    foo();
    foo();
    foo();
}
$ gcc -Wall -Werror static1.c
$ ./a.out
x=0, y=0
x=0, y=1
x=0, y=2
$
```

As you can see above, across different calls to `foo()`, the `static` variable `y` preserves its value. You must initialize a `static` variable to some value that can be assessed at compile time (like we did by assigning 0 to `y`), and that value will only be assigned once, when the program begins.

Later, you will see that `static` has a different meaning when used *on* a function, e.g. `static void foo() { ... }`. When we get to C++, you will encounter a *third* somewhat distinct use of the `static` keyword. Do also note that there is a difference between a statically allocated array and an array that is marked with the `static` keyword. In fact, a static pointer (e.g. `static int* ptr;`) can reference dynamically allocated memory.

5.2 File I/O

You will do file I/O in this programming assignment. We might talk more about file I/O during lecture, but below are enough details to get you through this assignment.

You should have learned about file I/O in ECS 32A and written programs that read from or wrote to files in ECS 32B. There is not much difference between how file I/O is done in Python vs. C. In both languages, we call a function to open a file, and we specify a certain mode. In the case of C, that function is `fopen()`[1]. The `fopen()` function returns `NULL` if it cannot open the file. Below are modes that you might find useful when dealing with text files in C. I think most of these are the same in Python.

- `"r"`: open existing[2] text file for reading.
- `"w"`: open/create file for writing, clearing its contents first.
- `"a"`: open/create file for writing at end.
- `"r+"`: open for reading and writing.

After we have successfully opened a file we are given a pointer to an instance of `FILE`, which is a struct[3] (see the next section for a description on structs), and we can use this file pointer in the same way in which we would use a "file object" in Python. We can then use any of the various I/O functions declared in `<stdio.h>` to read from the file, such as `fscanf()` or `fgets()`. (You can look up others.) As with Python, we still have to close the file when we are done with it.

Below is an example in which we use `fgets()` to read a file line-by-line, taking advantage of the fact that `fgets()` returns `NULL` – which, as you should recall, is a false value – once there is no more to read.

```
$ cat poem.txt
There once was a man from Peru;
who dreamed he was eating his shoe.
He woke with a fright
in the middle of the night
to find that his dream had come true.
```

[1] There is the `open()` function in the `<fnctl.h>` header, for compilers that support the C POSIX library, but we won't talk about that here. The file I/O functions in the C POSIX Library make it easier to access features specific to any filesystem (such as the filesystems supported by most Linux distributions) that is POSIX-compliant, such as directories, file permissions, and soft/hard links. The file I/O functions in `<stdio.h>`, such as `fopen()`, do not make this easy, since these functions – being part of the C standard – have to support filesystems that are not POSIX-compliant.

[2] Fails if file doesn't exist, just like how Python's `open()` function raises a `FileNotFoundError` exception if you try to open a nonexistent file.

[3] The reason you can say `FILE` instead of `struct FILE` is due to something called `typedef`, which we'll talk about during lecture.

```
$ cat files1.c
#include <stdio.h>

// Effectively means 98, because of the newline character (which fgets() reads)
// and the null terminator.
#define MAX_LINE_SIZE 100

int main()
{
    FILE* fp = fopen("poem.txt", "r");
    char buf[MAX_LINE_SIZE];
    int lineCounter = 0;
    while (fgets(buf, MAX_LINE_SIZE, fp))
    {
        // No newline character at end of format string because fgets() already
        // put the newline character at the end of buf.
        printf("Line #%d: %s", lineCounter, buf);
        lineCounter += 1;
    }
    fclose(fp);
    return 0;
}
$ gcc -Wall -Werror files1.c
$ ./a.out
Line #0: There once was a man from Peru;
Line #1: who dreamed he was eating his shoe.
Line #2: He woke with a fright
Line #3: in the middle of the night
Line #4: to find that his dream had come true.
$
```

Below is an example in which we use `fscanf()`. `fscanf()` is probably more helpful when we know the file will be formatted in a certain way, but this may not occur often. Sometimes, even when a text file has a certain format, you may prefer to use `fgets()` and then parse the line that was read, e.g. with `strtok()` and type conversion functions such as `atoi()` or `sscanf()`. It's all subjective, but in many cases, you probably can't go wrong with either.

```
$ cat values.txt
  58
abc  8.3

  & *
678
$ cat files2.c
#include <stdio.h>

#define BUF_LEN 15

int main()
{
    FILE* fp = fopen("values.txt", "r");
    int x, numRead;
    float y;
    char z, buf[BUF_LEN];
    numRead = fscanf(fp, "%d %s %f", &x, buf, &y);
    printf("numRead: %d\n", numRead);
    printf("x: %d\n", x);
    printf("buf: %s\n", buf);
    printf("y: %f\n", y);
    printf("===\n");
    // Note leading whitespace in format string
    // since reading character and want to ignore
    // whitespace.
    numRead = fscanf(fp, " %c", &z);
    printf("numRead: %d\n", numRead);
    printf("z: %c\n", z);
    printf("===\n");
    numRead = fscanf(fp, "%s", buf);
    printf("numRead: %d\n", numRead);
    printf("buf: %s\n", buf);
    printf("===\n");
    // Leading whitespace in format string
    // for reading the character.
    numRead = fscanf(fp, " %c %d", &z, &x);
    printf("numRead: %d\n", numRead);
```

```
    printf("z: %c\n", z);
    printf("x: %d\n", x);
    printf("===\n");
    // fscanf() returns EOF when end-of-file is reached.
    // EOF is likely -1, but you should not depend on that and should
    // instead use EOF.
    // fscanf() returns 0 if it tries to read a value that does not match
    // the format specifier, but that is different from reaching the end
    // of the file.
    numRead = fscanf(fp, "%d", &x);  // no more to read
    printf("numRead: %d\n", numRead);
    fclose(fp);
}
$ gcc -Wall -Werror files2.c
$ ./a.out
numRead: 3
x: 58
buf: abc
y: 8.300000
===
numRead: 1
z: &
===
numRead: 1
buf: *
===
numRead: 2
z: 6
x: 78
===
numRead: -1
$
```

The example below shows that `fscanf()` ignores whitespace in the format string, except to use whitespace as a separator. As was the case with `scanf()`, `fscanf()` is probably not as convenient as `fgets()` when trying to read a string that consists of multiple words.

```
$ cat goo.txt
what
       89
 22
$ cat files3.c
#include <stdio.h>

#define BUF_LEN 15

int main()
{
    FILE* fp = fopen("goo.txt", "r");
    char buf[BUF_LEN];
    int a, b;
    fscanf(fp, "%s %d %d", buf, &a, &b);
    printf("s=%s, a=%d, b=%d\n", buf, a, b);
    fclose(fp);
    return 0;
}
$ gcc -Wall -Werror files3.c
$ ./a.out
s=what, a=89, b=22
$
```

Writing to a file can be done with `fprintf()`, `fputs()`, or other functions that you can look up. Below is an example.

```
$ ls tmp
$ cat files4.c
#include <stdio.h>

int main()
{
    FILE* fp = fopen("tmp/output", "w");
    fputs("Hi there\n", fp);
    fprintf(fp, "%s %d %f\n", "blah", 53, 8.2);
    fclose(fp);
}
```

```
$ gcc -Wall -Werror files4.c
$ ./a.out
$ cat tmp/output
Hi there
blah 53 8.200000
$
```

5.3 Structs

You will use structs in this programming assignment. We might talk more about structs during lecture, but below are enough details to get you through this assignment.

You should have learned about classes in Python in ECS 32A and/or ECS 32B. In C, classes are instead called structs, and there are no methods. The type of each member must be specified. There are no initializers[4] in C, but when defining an instance of a struct, you can use an *initializer list* in which you specify the values of the members in the *same order* in which they appear in the definition of the struct. As is the case with dynamically allocated arrays, you cannot use an initializer list to initialize the members of a dynamically allocated instance of a struct. Below is an example in which we define a struct (`struct Person`) and create a few instances (`p1`, `p2`) of that struct.

```
$ cat person1.c
#include <stdio.h>

struct Person
{
    int age;
    char* firstName;
    char* lastName;
};

void printFullName(struct Person p)
{
    printf("Full name: %s %s\n", p.firstName, p.lastName);
}

int main()
{
    struct Person p1 = {35, "John", "Johnson"};  // first instance
    printf("%d %s %s\n", p1.age, p1.firstName, p1.lastName);
    printFullName(p1);
    struct Person p2 = {48, "Rich", "Richards"}; // second instance
    printFullName(p2);
}
$ gcc -Wall -Werror person1.c -o person1
$ ./person1
35 John Johnson
Full name: John Johnson
Full name: Rich Richards
$
```

As you can see, we can still use a struct to package data together. Although we cannot have methods, we can still have functions that act on instances of `struct Person`, such as `printFullName()`.

Typically, if a function takes an instance of a struct as an argument, then it is preferable to take a pointer to that instance instead. One purpose of this is that if you don't pass a pointer and instead directly pass the instance – as we did in the example above – then a *copy* of the instance is made, and the function will be passed this copy (recall `pass-by-value` from slide deck #5). This is not important when dealing with arguments of small types such as `int` or `char`, but structs can get large, and the time spent copying (and the unnecessary wasted space taken by the copy) may be considerable in certain applications. Below is an example in which we pass the address of the struct as argument instead.

```
$ cat person2.c
...
void printFullName(struct Person* p)
{
    printf("Full name: %s %s\n", (*p).firstName, (*p).lastName);
}

int main()
{
    struct Person p1 = {35, "John", "Johnson"};
```

[4] In C++, these are pretty much called constructors. I say "pretty much" because I believe that there is a subtle, insignificant difference that you can look up.

```
    printf("%d %s %s\n", p1.age, p1.firstName, p1.lastName);
    printFullName(&p1);
    struct Person p2 = {48, "Rich", "Richards"};
    printFullName(&p2);
}
$ gcc -Wall -Werror person2.c -o person2
$ ./person2
35 John Johnson
Full name: John Johnson
Full name: Rich Richards
$
```

Notice that in order to access the `firstName` and `lastName` members through the pointer `p` in `printFullName()`, we had to dereference the pointers first.[5] There is a shorthand for doing both a dereference and a member access simultaneously, as shown below:

```
// As seen above.
printf("Full name: %s %s\n", (*p).firstName, (*p).lastName);

// Same line, except using the shorthand.
printf("Full name: %s %s\n", p->firstName, p->lastName);
```

6 Programming Problems

As was the case with the first two parts of the last programming assignment, the first two parts of this assignment are intended to ease you into recent concepts that you have learned about, whether during lecture or in the sections above.

6.1 Restrictions

Since you will only submit `prog3.c` to the autograder, **you are not allowed to modify `prog3.h`.**
Below is a list of headers that you *are* allowed to include in `prog3.c`. You may not need all of these.

- `prog3.h`
  - You should not paste the definition of `struct Student` into `prog3.c`; including `prog3.h` takes care of this.
- `<stdio.h>`
- `<stdlib.h>`
- `<string.h>`
- `<limits.h>`
- `<stdbool.h>`
- `<ctype.h>`

The autograder will penalize you if your code has memory leaks or fails to close any file that it opens.

6.2 Part #1

In `prog3.c`, implement the `parseForHighest()` function that is declared in `prog3.h`. The first argument that this function takes is the name of a file that is assumed to contain one integer per line. (You don't need to validate that it contains one integer per line; just assume it.) The function should determine the highest integer in the file and store that integer in the `int` variable referenced by the second argument.

You need not worry about overflow or underflow, so long as you use the `int` type instead of something like `short`. Don't forget to close the file when you are done with it.

Input validation: Return -1 if either argument is a null pointer. Return -2 if the file cannot be opened.

Below are examples of how your program should behave.

```
$ cat test_parseForHighest.c
#include "prog3.h"

#include <stdio.h>

int main()
{
    int highest = 0, retval = 0;
```

[5] You need the parentheses around the `*p` (e.g. when doing `(*p).firstName`) because the dereference operator `*` has lower precedence than the member access operator `.`.

```
    retval = parseForHighest("values1.txt", &highest);
    printf("retval: %d\n", retval);
    printf("highest: %d\n", highest);
    retval = parseForHighest(NULL, &highest);
    printf("retval: %d\n", retval);
    retval = parseForHighest("values1.txt", NULL);
    printf("retval: %d\n", retval);
}
$ cat values1.txt
18
34
6
-3
5
$ gcc -Wall -Werror test_parseForHighest.c prog3.c -o test_parseForHighest
$ ./test_parseForHighest
retval: 0
highest: 34
retval: -1
retval: -1
$
```

6.3 Part #2

In `prog3.c`, implement the `getAllHigherThan()` function that is declared in `prog3.h`. This function takes four arguments: an array of integers, the length of that array, a threshold integer, and a pointer to an int. `getAllHigherThan()` should return an array of all integers in the first argument that are higher than the given threshold. Just for fun, the integers in this new array must appear in the opposite order compared to the order in which they appear in the original array. The length of the new array must be stored in the integer referenced by the fourth argument.

I do not plan on having the autograder penalize you if you use more heap memory than you need (although the autograder will check if you have memory leaks, as stated above). The autograder will call `free()` on the array that your function returns.

Input validation: Return `NULL` if the function is given any null pointers. Note that you cannot validate that the given length is "actually" the length of the array. (Sometimes, students ask about that.)

Below are examples of how your program should behave. Regarding the output of `valgrind`, you don't need to worry about details such as the values between the equal signs or the number of bytes allocated in total; the autograder will simply use `valgrind` to check for memory leaks.

```
$ cat test_getAllHigherThan.c
#include "prog3.h"

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int arr[] = {18, 12, 22, 37, 15};
    unsigned newArrlen = 0;
    int* higherThan = getAllHigherThan(arr, 5, 15, &newArrlen);
    for (unsigned i = 0; i < newArrlen; ++i)
        printf("Index %u: %d\n", i, higherThan[i]);
    free(higherThan);
    printf("===\n");
    int arr2[] = {38, 15, 16, 22, 9, 32, 25, 20};
    higherThan = getAllHigherThan(arr2, 8, 19, &newArrlen);
    for (unsigned i = 0; i < newArrlen; ++i)
        printf("Index %u: %d\n", i, higherThan[i]);
    free(higherThan);
}
$ gcc -Wall -Werror test_getAllHigherThan.c prog3.c -o test_getAllHigherThan
$ ./test_getAllHigherThan
Index 0: 37
Index 1: 22
Index 2: 18
===
Index 0: 20
Index 1: 25
Index 2: 32
Index 3: 22
Index 4: 38
$ valgrind ./test_getAllHigherThan
```

```
==13995== Memcheck, a memory error detector
==13995== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13995== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==13995== Command: ./test_getAllHigherThan
==13995==
Index 0: 37
Index 1: 22
Index 2: 18
===
Index 0: 20
Index 1: 25
Index 2: 32
Index 3: 22
Index 4: 38
==13995==
==13995== HEAP SUMMARY:
==13995==     in use at exit: 0 bytes in 0 blocks
==13995==   total heap usage: 3 allocs, 3 frees, 1,056 bytes allocated
==13995==
==13995== All heap blocks were freed -- no leaks are possible
==13995==
==13995== For counts of detected and suppressed errors, rerun with: -v
==13995== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
$
```

6.4 Part #3

In `prog3.c`, implement the `strtok_c()` function that is declared in `prog3.h`. This function takes the same arguments as `strtok()`. The difference is that the first argument is `const`, meaning that you are not allowed to modify that string. (Although there may be ways to get around `const` in C, the autograder will check if the string is modified by your `strtok_c()` function.) Rather than return a pointer to where the token starts in the original string like `strtok()` does, `strtok_c()` will instead return a *copy* of the token. As with `strtok()`, if the first argument is `NULL`, then `strtok_c()` should continue tokenizing from where it left off on the previous string. If there are no more tokens, then `strtok_c()` should return `NULL`.

As we saw a bit of during lecture, `strtok()` has some very specific behaviors in certain situations. Below are two very specific behaviors for `strtok_c()`, one of which works differently for `strtok()`.

1. **No empty tokens. Either a non-empty token is returned, or `NULL` is returned to indicate that there are no more tokens.**[6] As with `strtok()` (see example #3 on slide #50 of the slide deck about pointers), if the tokenizing were to start at a delimiter, `strtok_c()` should try to start at the next character instead, repeating until it starts at a character that is not a delimiter. (If the end of the string were reached during this process, then that would mean that there are no more tokens.)
2. Suppose you reach the end of the string in a given call to `strtok_c()`. If the next call to `strtok_c()` has `NULL` as the first argument, then you should return `NULL`. This might seem obvious and unimportant, but you can modify the string that you are tokenizing in between calls to `strtok_c()`, so there are scenarios in which if I didn't say what I just said, there would be ambiguity as to how to address those scenarios. For example, if I do `char s[7] = "abcde"; char* tok = strtok_c(s,"x");` followed by `s[5] = 'f'; char* tok2 = strtok_c(NULL,"x");`, then `tok2` should be `NULL`, not `"f"`. (With the normal `strtok()`, it would be `"f"`, but you can disregard this.)
3. If the first call that is ever done to `strtok_c()` has `NULL` as the first argument, then the function should return `NULL`.

Compared to the other parts on this assignment, the penalty for having memory leaks will be much higher on this part. You should think carefully about how to do this part before you start coding much of it. The autograder will call `free()` on whatever string is returned by your function.

Below are many examples of how your function should behave.

```
$ cat test_strtok_c.c
#include "prog3.h"

#include <stdio.h>
#include <stdlib.h>

#define NULL_PTR_STR "(null pointer)"

int main()
{
```

[6] While we're at it, I should point out that there is a distinction between a null pointer and an empty string (like `""`). A null pointer has the value `NULL`; an empty string does not. An empty string is a `char` array with one value: the null terminator.

```
    /**
     * Simple example.
     */
    char s1[] = "Hi there friend";  // Could use char* since the string
                                    // shouldn't be mutated anyways.
    char* tok1 = strtok_c(s1, " ");
    char* tok2 = strtok_c(NULL, " ");
    char* tok3 = strtok_c(NULL, " ");
    char* tok4 = strtok_c(NULL, " ");
    printf("s1: %s\n", s1);
    printf("tok1: %s\n", tok1);
    printf("tok2: %s\n", tok2);
    printf("tok3: %s\n", tok3);
    printf("tok4: %s\n", tok4 ? tok4 : NULL_PTR_STR);
    free(tok1);
    free(tok2);
    free(tok3);

    /**
     * Multiple delimiters.
     */
    char s2[] = "abracadabra";
    tok1 = strtok_c(s2, "db");
    tok2 = strtok_c(NULL, "db");
    tok3 = strtok_c(NULL, "db");
    tok4 = strtok_c(NULL, "db");
    char* tok5 = strtok_c(NULL, "db");
    printf("s2: %s\n", s2);
    printf("tok1: %s\n", tok1);
    printf("tok2: %s\n", tok2);
    printf("tok3: %s\n", tok3);
    printf("tok4: %s\n", tok4);
    printf("tok5: %s\n", tok5 ? tok5 : NULL_PTR_STR);
    free(tok1);
    free(tok2);
    free(tok3);
    free(tok4);

    /**
     * Tokenizing same string multiple times.
     */
    char s3[] = "How are you?";
    tok1 = strtok_c(s3, "er");
    tok2 = strtok_c(NULL, "o");  // Starts at 'e', but if we had used the
                                 // same delimiters as before, it would start
                                 // at ' '.
    tok3 = strtok_c(s3, "w?");
    printf("s3: %s\n", s3);
    printf("tok1: %s\n", tok1);
    printf("tok2: %s\n", tok2);
    printf("tok3: %s\n", tok3);
    free(tok1);
    free(tok2);
    free(tok3);

    /**
     * This is similar to the example I mentioned where I talk about no empty
     * tokens in the directions.
     */
    char s4[8] = "abcde";  // must be char[] since mutating two lines later
    tok1 = strtok_c(s4, "x");
    s4[5] = 'f';
    s4[6] = 'g';  // s4[7] is already a null byte
    tok2 = strtok_c(NULL, "x");  // Will return NULL since last strtok_c() call
                                 // reached what was the end of s4 at the time.
    printf("s4: %s\n", s4);
    printf("tok1: %s\n", tok1);
    printf("tok2: %s\n", tok2 ? tok2 : NULL_PTR_STR);
    free(tok1);
    free(tok2);

    /**
     * Example in which the tokens have to have their start points moved up
     * so as to avoid starting on delimiters.
```

```
     */
    char s5[] = "aabcdaaaefghaaiaaajklaa";
    tok1 = strtok_c(s5, "?!a");
    tok2 = strtok_c(NULL, "?!a");
    tok3 = strtok_c(NULL, "?!a");
    tok4 = strtok_c(NULL, "?!a");
    tok5 = strtok_c(NULL, "?!a");
    char* tok6 = strtok_c(NULL, "?!a");
    printf("s5: %s\n", s5);
    printf("tok1: %s\n", tok1);
    printf("tok2: %s\n", tok2);
    printf("tok3: %s\n", tok3);
    printf("tok4: %s\n", tok4);
    printf("tok5: %s\n", tok5 ? tok5 : NULL_PTR_STR);
    printf("tok6: %s\n", tok6 ? tok6 : NULL_PTR_STR);
    free(tok1);
    free(tok2);
    free(tok3);
    free(tok4);
}
$ gcc -Wall -Werror test_strtok_c.c prog3.c -o test_strtok_c
$ ./test_strtok_c
s1: Hi there friend
tok1: Hi
tok2: there
tok3: friend
tok4: (null pointer)
s2: abracadabra
tok1: a
tok2: raca
tok3: a
tok4: ra
tok5: (null pointer)
s3: How are you?
tok1: How a
tok2: e y
tok3: Ho
s4: abcdefg
tok1: abcde
tok2: (null pointer)
s5: aabcdaaaefghaaiaaajklaa
tok1: bcd
tok2: efgh
tok3: i
tok4: jkl
tok5: (null pointer)
tok6: (null pointer)
$ valgrind ./test_strtok_c
...
==20308==
==20308== HEAP SUMMARY:
==20308==     in use at exit: 0 bytes in 0 blocks
==20308==   total heap usage: 16 allocs, 16 frees, 1,086 bytes allocated
==20308==
==20308== All heap blocks were freed -- no leaks are possible
==20308==
==20308== For counts of detected and suppressed errors, rerun with: -v
==20308== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
$
```

6.5 Part #4

Below is the definition of `struct Student` from `prog3.h`. This struct is used to represent a college student. The `name` member is the full name of the student. The `currCourses` member is an array of strings containing the courses that the student is taking this quarter. The `numCurrCourses` member is the length of that array. The `prevCourses` member is an array of strings containing the courses that the student *has taken* before this quarter. The `numPrevCourses` member is the length of that array.

```
struct Student
{
    char* name;
    unsigned numCurrCourses;
    char** currCourses;
```

```
    unsigned numPrevCourses;
    char** prevCourses;
};
```

As you can see in `prog3.h`, in this part, you will implement four functions associated with the above struct. They are described below.

- `struct Student* loadStudent(const char* studentFilename)`: This function uses the named file to create an instance of `struct Student` and return (a pointer to) that instance. The function should return `NULL` if a null pointer was given or if the file could not be opened. The file contains all of the information needed to create an instance of `struct Student`. You can find examples in the `student_files` folder on Canvas. Each such file will *always* follow the following format with the order below (and you can assume this in your program):
  - The first line will be the name of the student.
  - The second line will be the number of courses that the student is taking this quarter.
  - Until a blank line is encountered, each line contains the name of a course that the student is taking this quarter. *There will always be at least one course here.*
  - *There will always be one completely blank line.*
  - The first line after the blank line will contain the number of courses that the student previously took.
  - Each line after this will contain the name of a course that the student took in a past quarter. *There will always be at least one course here.*
- `void printStudent(const struct Student* s)`: This prints all of the stored information about the student. See the examples below. The function should print nothing and avoid crashing if the given pointer is a null pointer.
- `void freeStudent(struct Student** s)`: This function frees all dynamically allocated memory associated with the instance of `struct Student` that is referenced by `s`. The function then sets the pointer referenced by `s` to `NULL` (which is why `s` is a double pointer). If `s` is a null pointer or if `s` references a null pointer, then the function should return immediately.
- `int areClassmates(const struct Student* s1, const struct Student* s2)`: This boolean function returns a true value if the given two students have any courses in common that they are currently taking. Otherwise, the function returns a false value, i.e. 0.

In `prog3.h`, there are two macros `MAX_LINE_LEN` and `BUF_LEN`. No line in the file whose name is given to `loadStudent()` will ever have more than 40 visible characters, and 40 is the value of `MAX_LINE_LEN`. If you choose to use `fgets()`, you may find `BUF_LEN` helpful.

Below are examples of how your code should behave.

```
$ cat test_student.c
#include "prog3.h"

#include <stdio.h>
#include <stdlib.h>

void reportIfClassmates(const struct Student *s1,
                        const struct Student *s2)
{
    char* tmp = "";
    if (!areClassmates(s1, s2))
        tmp = "not ";
    printf("%s and %s are %sclassmates.\n", s1->name, s2->name, tmp);
}

int main(int argc, char* argv[])
{
    if (argc != 4)
    {
        fprintf(stderr,
                "%s requires three file names as command-line arguments.\n",
                argv[0]);
        exit(1);
    }
    struct Student* s1 = loadStudent(argv[1]);
    printStudent(s1);
    struct Student* s2 = loadStudent(argv[2]);
    printStudent(s2);
    reportIfClassmates(s1, s2);
    struct Student* s3 = loadStudent(argv[3]);
    printStudent(s3);
    reportIfClassmates(s1, s3);
    reportIfClassmates(s2, s3);
```

```
    freeStudent(&s1);
    freeStudent(&s2);
    freeStudent(&s3);
}
$ gcc -Wall -Werror test_student.c prog3.c -o test_student
$ cat student_files/aaron_w20.txt
Grad Aaron
2
ECS 201A
ECS 222A

3
ECS 240
ECS 251
ECS 252A
$ cat student_files/aaron_s16.txt
Young Aaron
3
ECS 60
PHY 9A
MAT 22B

8
ECS 30
ECS 40
ECS 20
MAT 21C
MAT 21D
MAT 22A
CHE 2A
ENL 5F
$ cat student_files/jake_jacobs.txt
Jake Jacobs
4
ECS 36B
PHY 9A
ECS 20
MAT 21C

3
ECS 36A
MAT 21B
MAT 21A
$ ./test_student student_files/aaron_w20.txt  student_files/aaron_s16.txt student_files/jake_jacobs.txt
Name: Grad Aaron
Current courses:
ECS 201A
ECS 222A
Previous courses:
ECS 240
ECS 251
ECS 252A
Name: Young Aaron
Current courses:
ECS 60
PHY 9A
MAT 22B
Previous courses:
ECS 30
ECS 40
ECS 20
MAT 21C
MAT 21D
MAT 22A
CHE 2A
ENL 5F
Grad Aaron and Young Aaron are not classmates.
Name: Jake Jacobs
Current courses:
ECS 36B
PHY 9A
ECS 20
MAT 21C
Previous courses:
```

```
ECS 36A
MAT 21B
MAT 21A
Grad Aaron and Jake Jacobs are not classmates.
Young Aaron and Jake Jacobs are classmates.
$ valgrind ./test_student student_files/aaron_w20.txt  student_files/aaron_s16.txt student_files/
    jake_jacobs.txt
...
==20435==
==20435== HEAP SUMMARY:
==20435==     in use at exit: 0 bytes in 0 blocks
==20435==   total heap usage: 42 allocs, 42 frees, 15,485 bytes allocated
==20435==
==20435== All heap blocks were freed -- no leaks are possible
==20435==
==20435== For counts of detected and suppressed errors, rerun with: -v
==20435== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
$
```

UCDAVIS
COMPUTER SCIENCE