

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` 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`, 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.

```

1 $ cat static1.c
2 #include <stdio.h>
3
4 void foo(void)
5 {
6     int x = 0;
7     static int y = 0;
8     printf("x=%d, y=%d\n", x, y);
9     x += 1;
10    y += 1;
11 }
12
13 int main()
14 {
15     foo();
16     foo();
17     foo();
18 }
19 $ gcc -Wall -Werror static1.c
20 $ ./a.out
21 x=0, y=0
22 x=0, y=1
23 x=0, y=2
24 $

```

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 existing² 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 struct³ (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.

```

1 $ cat poem.txt
2 There once was a man from Peru;
3 who dreamed he was eating his shoe.
4 He woke with a fright
5 in the middle of the night
6 to find that his dream had come true.

```

¹There is the `open()` function in the `<fnct1.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.

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

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

```

7 $ cat files1.c
8 #include <stdio.h>
9
10 // Effectively means 98, because of the newline character (which fgets() reads)
11 // and the null terminator.
12 #define MAX_LINE_SIZE 100
13
14 int main()
15 {
16     FILE* fp = fopen("poem.txt", "r");
17     char buf[MAX_LINE_SIZE];
18     int lineCounter = 0;
19     while (fgets(buf, MAX_LINE_SIZE, fp))
20     {
21         // No newline character at end of format string because fgets() already
22         // put the newline character at the end of buf.
23         printf("Line #%d: %s", lineCounter, buf);
24         lineCounter += 1;
25     }
26     fclose(fp);
27     return 0;
28 }
29 $ gcc -Wall -Werror files1.c
30 $ ./a.out
31 Line #0: There once was a man from Peru;
32 Line #1: who dreamed he was eating his shoe.
33 Line #2: He woke with a fright
34 Line #3: in the middle of the night
35 Line #4: to find that his dream had come true.
36 $

```

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.

```

1 $ cat values.txt
2 58
3 abc 8.3
4
5 & *
6 678
7 $ cat files2.c
8 #include <stdio.h>
9
10 #define BUF_LEN 15
11
12 int main()
13 {
14     FILE* fp = fopen("values.txt", "r");
15     int x, numRead;
16     float y;
17     char z, buf[BUF_LEN];
18     numRead = fscanf(fp, "%d %s %f", &x, buf, &y);
19     printf("numRead: %d\n", numRead);
20     printf("x: %d\n", x);
21     printf("buf: %s\n", buf);
22     printf("y: %f\n", y);
23     printf("===\n");
24     // Note leading whitespace in format string
25     // since reading character and want to ignore
26     // whitespace.
27     numRead = fscanf(fp, " %c", &z);
28     printf("numRead: %d\n", numRead);
29     printf("z: %c\n", z);
30     printf("===\n");
31     numRead = fscanf(fp, "%s", buf);
32     printf("numRead: %d\n", numRead);
33     printf("buf: %s\n", buf);
34     printf("===\n");
35     // Leading whitespace in format string
36     // for reading the character.
37     numRead = fscanf(fp, " %c %d", &z, &x);
38     printf("numRead: %d\n", numRead);

```

```

39     printf("z: %c\n", z);
40     printf("x: %d\n", x);
41     printf("===\n");
42     // fscanf() returns EOF when end-of-file is reached.
43     // EOF is likely -1, but you should not depend on that and should
44     // instead use EOF.
45     // fscanf() returns 0 if it tries to read a value that does not match
46     // the format specifier, but that is different from reaching the end
47     // of the file.
48     numRead = fscanf(fp, "%d", &x); // no more to read
49     printf("numRead: %d\n", numRead);
50     fclose(fp);
51 }
52 $ gcc -Wall -Werror files2.c
53 $ ./a.out
54 numRead: 3
55 x: 58
56 buf: abc
57 y: 8.300000
58 ===
59 numRead: 1
60 z: &
61 ===
62 numRead: 1
63 buf: *
64 ===
65 numRead: 2
66 z: 6
67 x: 78
68 ===
69 numRead: -1
70 $

```

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.

```

1 $ cat goo.txt
2 what
3     89
4     22
5 $ cat files3.c
6 #include <stdio.h>
7
8 #define BUF_LEN 15
9
10 int main()
11 {
12     FILE* fp = fopen("goo.txt", "r");
13     char buf[BUF_LEN];
14     int a, b;
15     fscanf(fp, "%s %d %d", buf, &a, &b);
16     printf("s=%s, a=%d, b=%d\n", buf, a, b);
17     fclose(fp);
18     return 0;
19 }
20 $ gcc -Wall -Werror files3.c
21 $ ./a.out
22 s=what, a=89, b=22
23 $

```

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

```

1 $ ls tmp
2 $ cat files4.c
3 #include <stdio.h>
4
5 int main()
6 {
7     FILE* fp = fopen("tmp/output", "w");
8     fputs("Hi there\n", fp);
9     fprintf(fp, "%s %d %f\n", "blah", 53, 8.2);
10    fclose(fp);
11 }

```

```

12 $ gcc -Wall -Werror files4.c
13 $ ./a.out
14 $ cat tmp/output
15 Hi there
16 blah 53 8.200000
17 $

```

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 initializers⁴ 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.

```

1 $ cat person1.c
2 #include <stdio.h>
3
4 struct Person
5 {
6     int age;
7     char* firstName;
8     char* lastName;
9 };
10
11 void printFullName(struct Person p)
12 {
13     printf("Full name: %s %s\n", p.firstName, p.lastName);
14 }
15
16 int main()
17 {
18     struct Person p1 = {35, "John", "Johnson"}; // first instance
19     printf("%d %s %s\n", p1.age, p1.firstName, p1.lastName);
20     printFullName(p1);
21     struct Person p2 = {48, "Rich", "Richards"}; // second instance
22     printFullName(p2);
23 }
24 $ gcc -Wall -Werror person1.c -o person1
25 $ ./person1
26 35 John Johnson
27 Full name: John Johnson
28 Full name: Rich Richards
29 $

```

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.

```

1 $ cat person2.c
2 ...
3 void printFullName(struct Person* p)
4 {
5     printf("Full name: %s %s\n", (*p).firstName, (*p).lastName);
6 }
7
8 int main()
9 {
10     struct Person p1 = {35, "John", "Johnson"};

```

⁴In 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.

```

11     printf("%d %s %s\n", p1.age, p1.firstName, p1.lastName);
12     printFullName(&p1);
13     struct Person p2 = {48, "Rich", "Richards"};
14     printFullName(&p2);
15 }
16 $ gcc -Wall -Werror person2.c -o person2
17 $ ./person2
18 35 John Johnson
19 Full name: John Johnson
20 Full name: Rich Richards
21 $

```

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:

```

1 // As seen above.
2 printf("Full name: %s %s\n", (*p).firstName, (*p).lastName);
3
4 // Same line, except using the shorthand.
5 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.

```

1 $ cat test_parseForHighest.c
2 #include "prog3.h"
3
4 #include <stdio.h>
5
6 int main()
7 {
8     int highest = 0, retval = 0;

```

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

```

9     retval = parseForHighest("values1.txt", &highest);
10    printf("retval: %d\n", retval);
11    printf("highest: %d\n", highest);
12    retval = parseForHighest(NULL, &highest);
13    printf("retval: %d\n", retval);
14    retval = parseForHighest("values1.txt", NULL);
15    printf("retval: %d\n", retval);
16 }
17 $ cat values1.txt
18 18
19 34
20 6
21 -3
22 5
23 $ gcc -Wall -Werror test_parseForHighest.c prog3.c -o test_parseForHighest
24 $ ./test_parseForHighest
25 retval: 0
26 highest: 34
27 retval: -1
28 retval: -1
29 $

```

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.

```

1 $ cat test_getAllHigherThan.c
2 #include "prog3.h"
3
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main()
8 {
9     int arr[] = {18, 12, 22, 37, 15};
10    unsigned newArrlen = 0;
11    int* higherThan = getAllHigherThan(arr, 5, 15, &newArrlen);
12    for (unsigned i = 0; i < newArrlen; ++i)
13        printf("Index %u: %d\n", i, higherThan[i]);
14    free(higherThan);
15    printf("===\n");
16    int arr2[] = {38, 15, 16, 22, 9, 32, 25, 20};
17    higherThan = getAllHigherThan(arr2, 8, 19, &newArrlen);
18    for (unsigned i = 0; i < newArrlen; ++i)
19        printf("Index %u: %d\n", i, higherThan[i]);
20    free(higherThan);
21 }
22 $ gcc -Wall -Werror test_getAllHigherThan.c prog3.c -o test_getAllHigherThan
23 $ ./test_getAllHigherThan
24 Index 0: 37
25 Index 1: 22
26 Index 2: 18
27 ===
28 Index 0: 20
29 Index 1: 25
30 Index 2: 32
31 Index 3: 22
32 Index 4: 38
33 $ valgrind ./test_getAllHigherThan

```



```

34 ==13995== Memcheck, a memory error detector
35 ==13995== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
36 ==13995== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
37 ==13995== Command: ./test_getAllHigherThan
38 ==13995==
39 Index 0: 37
40 Index 1: 22
41 Index 2: 18
42 ==
43 Index 0: 20
44 Index 1: 25
45 Index 2: 32
46 Index 3: 22
47 Index 4: 38
48 ==13995==
49 ==13995== HEAP SUMMARY:
50 ==13995==      in use at exit: 0 bytes in 0 blocks
51 ==13995==    total heap usage: 3 allocs, 3 frees, 1,056 bytes allocated
52 ==13995==
53 ==13995== All heap blocks were freed -- no leaks are possible
54 ==13995==
55 ==13995== For counts of detected and suppressed errors, rerun with: -v
56 ==13995== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
57 $

```

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.

```

1 $ cat test_strtok_c.c
2 #include "prog3.h"
3
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 #define NULL_PTR_STR "(null pointer)"
8
9 int main()
10 {

```

⁶While 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.

```

11  /**
12   * Simple example.
13   */
14   char s1[] = "Hi there friend"; // Could use char* since the string
15                                   // shouldn't be mutated anyways.
16   char* tok1 = strtok_c(s1, " ");
17   char* tok2 = strtok_c(NULL, " ");
18   char* tok3 = strtok_c(NULL, " ");
19   char* tok4 = strtok_c(NULL, " ");
20   printf("s1: %s\n", s1);
21   printf("tok1: %s\n", tok1);
22   printf("tok2: %s\n", tok2);
23   printf("tok3: %s\n", tok3);
24   printf("tok4: %s\n", tok4 ? tok4 : NULL_PTR_STR);
25   free(tok1);
26   free(tok2);
27   free(tok3);
28
29  /**
30   * Multiple delimiters.
31   */
32   char s2[] = "abracadabra";
33   tok1 = strtok_c(s2, "db");
34   tok2 = strtok_c(NULL, "db");
35   tok3 = strtok_c(NULL, "db");
36   tok4 = strtok_c(NULL, "db");
37   char* tok5 = strtok_c(NULL, "db");
38   printf("s2: %s\n", s2);
39   printf("tok1: %s\n", tok1);
40   printf("tok2: %s\n", tok2);
41   printf("tok3: %s\n", tok3);
42   printf("tok4: %s\n", tok4);
43   printf("tok5: %s\n", tok5 ? tok5 : NULL_PTR_STR);
44   free(tok1);
45   free(tok2);
46   free(tok3);
47   free(tok4);
48
49  /**
50   * Tokenizing same string multiple times.
51   */
52   char s3[] = "How are you?";
53   tok1 = strtok_c(s3, "er");
54   tok2 = strtok_c(NULL, "o"); // Starts at 'e', but if we had used the
55                                   // same delimiters as before, it would start
56                                   // at ' '.
57   tok3 = strtok_c(s3, "w?");
58   printf("s3: %s\n", s3);
59   printf("tok1: %s\n", tok1);
60   printf("tok2: %s\n", tok2);
61   printf("tok3: %s\n", tok3);
62   free(tok1);
63   free(tok2);
64   free(tok3);
65
66  /**
67   * This is similar to the example I mentioned where I talk about no empty
68   * tokens in the directions.
69   */
70   char s4[8] = "abcde"; // must be char[] since mutating two lines later
71   tok1 = strtok_c(s4, "x");
72   s4[5] = 'f';
73   s4[6] = 'g'; // s4[7] is already a null byte
74   tok2 = strtok_c(NULL, "x"); // Will return NULL since last strtok_c() call
75                                   // reached what was the end of s4 at the time.
76   printf("s4: %s\n", s4);
77   printf("tok1: %s\n", tok1);
78   printf("tok2: %s\n", tok2 ? tok2 : NULL_PTR_STR);
79   free(tok1);
80   free(tok2);
81
82  /**
83   * Example in which the tokens have to have their start points moved up
84   * so as to avoid starting on delimiters.

```

```

85     */
86     char s5[] = "aabcdaaaefghaaiaaajklaa";
87     tok1 = strtok_c(s5, "?!a");
88     tok2 = strtok_c(NULL, "?!a");
89     tok3 = strtok_c(NULL, "?!a");
90     tok4 = strtok_c(NULL, "?!a");
91     tok5 = strtok_c(NULL, "?!a");
92     char* tok6 = strtok_c(NULL, "?!a");
93     printf("s5: %s\n", s5);
94     printf("tok1: %s\n", tok1);
95     printf("tok2: %s\n", tok2);
96     printf("tok3: %s\n", tok3);
97     printf("tok4: %s\n", tok4);
98     printf("tok5: %s\n", tok5 ? tok5 : NULL_PTR_STR);
99     printf("tok6: %s\n", tok6 ? tok6 : NULL_PTR_STR);
100    free(tok1);
101    free(tok2);
102    free(tok3);
103    free(tok4);
104 }
105 $ gcc -Wall -Werror test_strtok_c.c prog3.c -o test_strtok_c
106 $ ./test_strtok_c
107 s1: Hi there friend
108 tok1: Hi
109 tok2: there
110 tok3: friend
111 tok4: (null pointer)
112 s2: abracadabra
113 tok1: a
114 tok2: raca
115 tok3: a
116 tok4: ra
117 tok5: (null pointer)
118 s3: How are you?
119 tok1: How a
120 tok2: e y
121 tok3: Ho
122 s4: abcdefg
123 tok1: abcde
124 tok2: (null pointer)
125 s5: aabcdaaaefghaaiaaajklaa
126 tok1: bcd
127 tok2: efgh
128 tok3: i
129 tok4: jkl
130 tok5: (null pointer)
131 tok6: (null pointer)
132 $ valgrind ./test_strtok_c
133 ...
134 ==20308==
135 ==20308== HEAP SUMMARY:
136 ==20308==      in use at exit: 0 bytes in 0 blocks
137 ==20308==    total heap usage: 16 allocs, 16 frees, 1,086 bytes allocated
138 ==20308==
139 ==20308== All heap blocks were freed -- no leaks are possible
140 ==20308==
141 ==20308== For counts of detected and suppressed errors, rerun with: -v
142 ==20308== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
143 $

```

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.

```

1 struct Student
2 {
3     char* name;
4     unsigned numCurrCourses;
5     char** currCourses;

```

```

6 unsigned numPrevCourses;
7 char** prevCourses;
8 };

```

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.

```

1 $ cat test_student.c
2 #include "prog3.h"
3
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 void reportIfClassmates(const struct Student *s1,
8                        const struct Student *s2)
9 {
10     char* tmp = "";
11     if (!areClassmates(s1, s2))
12         tmp = "not ";
13     printf("%s and %s are %sclassmates.\n", s1->name, s2->name, tmp);
14 }
15
16 int main(int argc, char* argv[])
17 {
18     if (argc != 4)
19     {
20         fprintf(stderr,
21             "%s requires three file names as command-line arguments.\n",
22             argv[0]);
23         exit(1);
24     }
25     struct Student* s1 = loadStudent(argv[1]);
26     printStudent(s1);
27     struct Student* s2 = loadStudent(argv[2]);
28     printStudent(s2);
29     reportIfClassmates(s1, s2);
30     struct Student* s3 = loadStudent(argv[3]);
31     printStudent(s3);
32     reportIfClassmates(s1, s3);
33     reportIfClassmates(s2, s3);

```

```

34     freeStudent(&s1);
35     freeStudent(&s2);
36     freeStudent(&s3);
37 }
38 $ gcc -Wall -Werror test_student.c prog3.c -o test_student
39 $ cat student_files/aaron_w20.txt
40 Grad Aaron
41 2
42 ECS 201A
43 ECS 222A
44
45 3
46 ECS 240
47 ECS 251
48 ECS 252A
49 $ cat student_files/aaron_s16.txt
50 Young Aaron
51 3
52 ECS 60
53 PHY 9A
54 MAT 22B
55
56 8
57 ECS 30
58 ECS 40
59 ECS 20
60 MAT 21C
61 MAT 21D
62 MAT 22A
63 CHE 2A
64 ENL 5F
65 $ cat student_files/jake_jacobs.txt
66 Jake Jacobs
67 4
68 ECS 36B
69 PHY 9A
70 ECS 20
71 MAT 21C
72
73 3
74 ECS 36A
75 MAT 21B
76 MAT 21A
77 $ ./test_student student_files/aaron_w20.txt student_files/aaron_s16.txt student_files/jake_jacobs.txt
78 Name: Grad Aaron
79 Current courses:
80 ECS 201A
81 ECS 222A
82 Previous courses:
83 ECS 240
84 ECS 251
85 ECS 252A
86 Name: Young Aaron
87 Current courses:
88 ECS 60
89 PHY 9A
90 MAT 22B
91 Previous courses:
92 ECS 30
93 ECS 40
94 ECS 20
95 MAT 21C
96 MAT 21D
97 MAT 22A
98 CHE 2A
99 ENL 5F
100 Grad Aaron and Young Aaron are not classmates.
101 Name: Jake Jacobs
102 Current courses:
103 ECS 36B
104 PHY 9A
105 ECS 20
106 MAT 21C
107 Previous courses:

```

```
108 ECS 36A
109 MAT 21B
110 MAT 21A
111 Grad Aaron and Jake Jacobs are not classmates.
112 Young Aaron and Jake Jacobs are classmates.
113 $ valgrind ./test_student student_files/aaron_w20.txt student_files/aaron_s16.txt student_files/
    jake_jacobs.txt
114 ...
115 ==20435==
116 ==20435== HEAP SUMMARY:
117 ==20435==      in use at exit: 0 bytes in 0 blocks
118 ==20435==    total heap usage: 42 allocs, 42 frees, 15,485 bytes allocated
119 ==20435==
120 ==20435== All heap blocks were freed -- no leaks are possible
121 ==20435==
122 ==20435== For counts of detected and suppressed errors, rerun with: -v
123 ==20435== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
124 $
```

