ECS 34: Programming Assignment #2

Instructor: Aaron Kaloti

Fall 2020

Contents

1 Changelog

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

- v.1: Initial version.
- v.2: Clarified that input validation is not required for the last part.
- v.3:
  - Added relative worth of each part in the Grading Breakdown section.
  - Added autograder details.
  - Added hint about using the `%` operator with a negative dividend to part #3.

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 Thursday, October 22. Gradescope will say 12:30 AM on Friday, October 23, 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.*

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 100 points. Here is the worth of each part.

- Part #1: 10
- Part #2: 10
- Part #3: 40
- Part #4: 20
- Part #5: 20

4 Submitting on Gradescope

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

4.1 Regarding Autograder

As a reminder, you will only submit `prog2.c`.

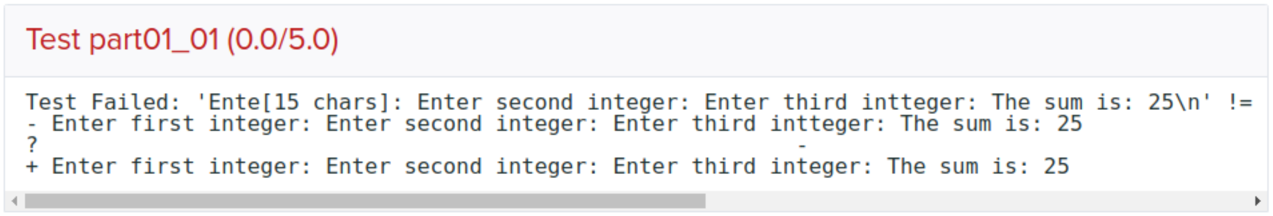
4.1.1 Interpreting Autograder's Messages on Gradescope

Your output must match mine *exactly*.

If your program crashes on a specific test case, it will be as if your program (compiled with the autograder) output nothing. (I hope to change this for the future autograders.) Thus, if it seems as if your program output nothing despite all of the `printf()` statements in `autograder_prog2.c`, this shouldn't be due to an autograder bug; your program simply crashed, e.g. a segmentation fault occurred.

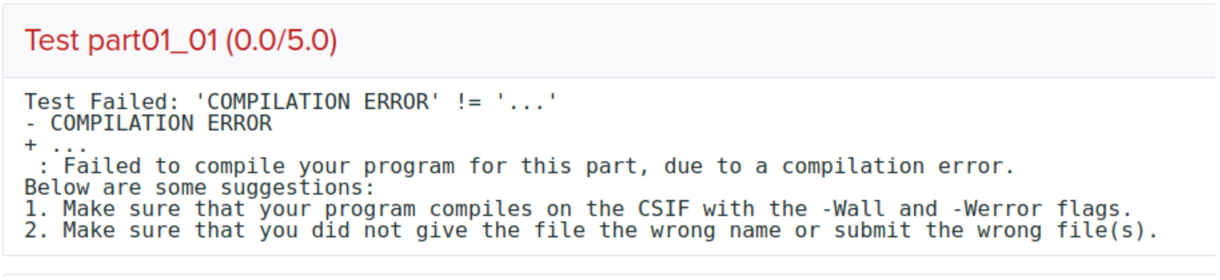
All of the rest of this subsection is from `prog_hw1.pdf`.

You will be told if you got the visible test cases correct or not a little bit after you submit to Gradescope (should not take more than 30 seconds, unless something is wrong with your programs). For each visible test case in this particular programming assignment (might not be true in future programming assignments), you will be told where the output differs. Below is an example. The first line of the output is probably useless; you should try to understand the rest of the output. Any line that begins with a `-` is a line that your program output that it was not supposed to; any line with a `+` is a line that your program *did not output* that it was supposed to. In computer science, it is common for software that finds the differences between two bodies of text (i.e. that "diffs" two bodies of text) to report it in this line-by-line fashion, so you should become accustomed to it. The line that starts with a question mark `?` later has a `-` in it (as you can see in the third line of the output below) that shows where the two lines begin to mismatch.


Test part01_01 (0.0/5.0)

```
Test Failed: 'Ente[15 chars]: Enter second integer: Enter third intteger: The sum is: 25\n' !=
- Enter first integer: Enter second integer: Enter third intteger: The sum is: 25
?                                                           -
+ Enter first integer: Enter second integer: Enter third integer: The sum is: 25
```

If either of your programs failed to compile, you will see a message like the below. Make sure to read the suggestions; students sometimes ignore those.


Test part01_01 (0.0/5.0)

```
Test Failed: 'COMPILATION ERROR' != '...'
- COMPILATION ERROR
+ ...
 : Failed to compile your program for this part, due to a compilation error.
Below are some suggestions:
1. Make sure that your program compiles on the CSIF with the -Wall and -Werror flags.
2. Make sure that you did not give the file the wrong name or submit the wrong file(s).
```

4.1.2 Test Cases Inputs

Number of visible vs. hidden: Here is the breakdown of visible and hidden test cases.

- Part #1: 3 cases (3 visible).
- Part #2: 3 cases (3 visible).
- Part #3: 16 cases (12 visible, 4 hidden).
- Part #4: 6 cases (3 visible, 3 hidden).
- Part #5: 4 cases (3 visible, 1 hidden).

Inputs: See `autograder_prog2.c`, `visible_cases.h`, and `visible_cases.c` on Canvas. To run the test cases on your end, follow the steps shown in the command line interaction below.

```
$ ls -1
autograder_prog2.c
prog2.c
prog2.h
visible_cases.c
visible_cases.h
$ gcc -Wall -Werror autograder_prog2.c prog2.c visible_cases.c -o autograder_prog2
$ ./autograder_prog2 1102   # part01_case01
s2=160
$ ./autograder_prog2 1303   # part03_case03
retval: 1
str: GHIP@ghip
$
```

When running the executable, you need to provide a special code after the executable's name, as shown in the end of the above command line interaction. The code is of the form 1YZZ, where Y indicates the part number, e.g. 4 means the case is for part 4. ZZ indicates the case number, e.g. 05 means case 05, 12 means case 12. Note that the parts after the `#` are not important; those are how to indicate comments on a command line.

5 Programming Problems

5.1 Files

In each part, you will implement a function(s). **All of your functions should go into a file called `prog2.c`.** Each function that you are required to define is already declared in `prog2.h`, so the functions as you define them in `prog2.c` must have the same signatures as they do in `prog2.h`. (I think you can change the names of the parameters.) **You cannot modify `prog2.h`. Even if you submit your own copy to the autograder, the autograder will ignore it.** `prog2.c` file should not contain a definition of `main()`. You may find it more convenient to put a `main()` in `prog2.c` when testing your code. If you do, just make sure to comment it out before submitting it to the autograder, or else the autograder will fail to compile your code. In the examples below, I demonstrate how to compile `prog2.c` with a different C file that contains the `main()` implementation.

5.2 Restrictions

In `prog2.c` (the only file that you are submitting), you are *only* allowed to include the following files:

- `prog2.h`
- `<stdio.h>`
- `<stdbool.h>`

That means that you *cannot* include `<stdlib.h>`, `<ctype.h>`, or `<string.h>`. **You may get a zero (or at least some sort of penalty) on this assignment if you violate this restriction.** As I advise below, you may find it useful to write your own implementations of some of the functions in `<ctype.h>` and/or `<string.h>`. If you do, make sure that the name isn't the same, e.g. if you write a function to get the length of a string, then you should name it `strLen()` instead of `strlen()`, or else you might get an odd compiler error, even if you do not include `<string.h>` (which you cannot do anyways).

5.3 Part #1: `sum3()`

Implement a function called `sum3` that takes four arguments: three integers and one pointer to an integer. (As mentioned above, you should refer to the signature in `prog2.h`.) This function should place the sum of the three integers into the variable referenced by the fourth argument.

Input validation: if the pointer given is `NULL`, then your function should do nothing (and avoid crashing).

Below are examples of how your program should behave on a Linux command line. Observe how I place a `main()` implementation in `test_sum.c` and compile `test_sum.c` and `prog2.c` together. As stated above, I recommend that you not have a `main()` implementation in your `prog2.c`. If you do, then you will get an error when you try to compile `test_sum.c` and `prog2.c` together, since there can only be one `main()`. Notice, too, that I do not mention `prog2.h` in the `gcc` command below. You should not mention header files in your `gcc` commands. They are taken care of when the `#include` directives are evaluated.

```
$ cat test_sum.c
#include "prog2.h"

#include <stdio.h>

int main()
{
    int s;
    sum3(4, 10, -1, &s);
    printf("%d\n", s);
    sum3(0, 2, -10, &s);
    printf("%d\n", s);
}
$ gcc -Wall -Werror test_sum.c prog2.c -o test_sum
$ ./test_sum
13
-8
$
```

5.4 Part #2: `replaceIfHigher()`

Implement a function called `replaceIfHigher()` that takes as arguments an array of `long` integers and the length of this array. (Once again, you should refer to the function declaration in `prog2.h`.) The function should prompt the user to enter a number of `long` integers equal to the size of the array. If the ith value that the user entered is greater than the ith value in the array, then the array's value should be replaced by what the user entered. For example, if the second value that the user entered is 100 and the second value (index 1) in the array is 47, then the array should be modified to have 100 at index 1.

Don't worry about arithmetic overflow.

Input validation: don't worry about it for this part.

Here are examples of how your program should behave on a Linux command line.

```
$ cat test_replaceIfHigher.c
#include "prog2.h"

#include <stdio.h>

int main()
{
    long vals[] = {583, 3000000000, 14, 6};
    replaceIfHigher(vals, 4);
    printf("=== vals ===\n");
    for (unsigned i = 0; i < 4; ++i)
        printf("vals[ %u ] is %ld\n", i, vals[i]);
}
$ gcc -Wall -Werror test_replaceIfHigher.c prog2.c -o test_replaceIfHigher
$ ./test_replaceIfHigher
1500
-3
20
3
=== vals ===
vals[ 0 ] is 1500
vals[ 1 ] is 3000000000
vals[ 2 ] is 20
vals[ 3 ] is 6
$ ./test_replaceIfHigher
-10
4000000000
20
0
=== vals ===
vals[ 0 ] is 583
vals[ 1 ] is 4000000000
vals[ 2 ] is 20
vals[ 3 ] is 6
$
```

5.5 Part #3: Shift Cipher

5.5.1 Brief Introduction to Cryptography and Shift Ciphers

Much of this contains exact wordings from, or paraphrasings of, parts of *Understanding Cryptography* by Christof Paar and Jan Pelzl.

Cryptography is the "science of secret writing with the goal of hiding the meaning of a message". This typically involves the processes of encryption and decryption, where **encryption** is the enciphering of the **plain text** into **cipher text** and **decryption** is the deciphering of the **cipher text** into the original **plain text**. Cryptography can be viewed as having three main branches:

- Symmetric Algorithms: two parties have an encryption method and a decryption method for which they share a secret key.
- Asymmetric (or Public-Key) Algorithms: a user possesses a secret key and a public key.
- Cryptographic Protocols: applications of cryptographic algorithms (e.g. the Transport Layer Security (TLS) scheme used in every web browser).

In this part, you will implement the shift/Caesar cipher, one of the most basic symmetric algorithms.

5.5.2 Directions

In this part, you will implement the shift/Caesar cipher by implementing the function `performShiftCipher()` that is declared in `prog2.h`. This function takes as argument a string and a key k. For each character in the string *that is a letter or digit*, the program should replace that character with whatever character comes k slots later in the ASCII table. If k is negative, then the shifting should go backwards. If the character is not a letter or digit, then it should not be changed.

If a character is shifted past the end of its "class", then a "wrap-around" should occur. For example, if $k = 4$, then "ZYzx96" should become "DCdb30", because:

- 'Z' would be shifted to 'D'.
- 'Y' would be shifted to 'C'.
- 'z' would be shifted to 'd'.
- 'x' would be shifted to 'b'.
- '9' would be shifted to '3'.
- '6' would be shifted to '0'.

Similar logic follows for if a character is shifted to being before the start of its "class". For example, if $k = -5$, then "CAa13" should become "XVv68", because:

- 'C' would be shifted to 'X'.
- 'A' would be shifted to 'V'.
- 'a' would be shifted to 'v'.
- '1' would be shifted to '6'.
- '3' would be shifted to '8'.

A character cannot be shifted into a different class, e.g. an uppercase letter cannot be shifted into being a lowercase letter and will remain an uppercase letter regardless of how much it is shited. Wrapping around can occur multiple times for any specific character, esp. if k is large, e.g. $k = 500$.

Input validation: if a null pointer is given, or if k is not in the range $[-500, 500]$, then your function should return a false value (i.e. 0). Otherwise, your function should return a true value (i.e. any nonzero value; the autograder won't care).

Note of caution: The shift cipher is a commonly referenced algorithm and commonly used homework assignment. You could easily find implementations of it online on the Internet. **Don't do that.** You should not look up code to copy online. Nor should you look up code online that you can just change slightly and submit as your own. Doing so may result in you being reported to the OSSJA.

Reminder: you are *not allowed* to include `<ctype.h>`. I agree that there are some useful functions in this header, e.g. a function for checking if a character is a letter; with what you have learned about the relationship between characters and integers, you should be able to write your own implementations of whatever functions from `<ctype.h>` that you might need. If you do write such functions, make sure that they do not have the same name as the `<ctype.h>` versions, or else you will get a compiler error about conflicting types.

Hint: If you use the `%` operator, you may encounter confusing behavior when the dividend is negative. I found this StackOverflow post helpful. If you happen to use any code from it, make sure to cite the source in a comment.

Here are examples of how your program should behave on a Linux command line. Note that if a text is encrypted with the shift cipher for a specific value of k, then you can decrypt it by running the shift cipher on the cipher text with the negative of that value of k.

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

#include "prog2.h"

int main()
{
    char str[] = "Hi there";
    int retval = performShiftCipher(str, 3);
    printf("str=%s, retval=%d\n", str, retval ? 1 : 0);
    retval = performShiftCipher(str, -3);
    printf("str=%s, retval=%d\n", str, retval ? 1 : 0);
    retval = performShiftCipher(NULL, 10);
    printf("str=%s, retval=%d\n", str, retval ? 1 : 0);
    retval = performShiftCipher(str, 500);
    printf("str=%s, retval=%d\n", str, retval ? 1 : 0);
    char str2[] = "56@$aBc* 4Zy";
    retval = performShiftCipher(str2, 9);
    printf("str2=%s, retval=%d\n", str2, retval ? 1 : 0);
    retval = performShiftCipher(str2, -9);
    printf("str2=%s, retval=%d\n", str2, retval ? 1 : 0);
}
$ gcc -Wall -Werror test_performShiftCipher.c prog2.c -o test_performShiftCipher
$ ./test_performShiftCipher
str=Kl wkhuh, retval=1
str=Hi there, retval=1
str=Hi there, retval=0
str=No znkxk, retval=1
str2=45@$jKl* 3Ih, retval=1
str2=56@$aBc* 4Zy, retval=1
$
```

5.6 Part #4: `strrstr()`

Recall from slide #53 (at the time of publishing of v.3) of slide deck #6 that `strstr()` returns a pointer to the *first* occurrence of the second string in the first string. In this part, you will implement the reverse version, `strrstr()`. This function returns a pointer to the *last* occurrence of the second string in the first string, or `NULL` if there are no occurrences. Note that you are not returning a copy of either string (or of a part of either string); no copying should be done.

Reminder: you are *not allowed* to include `<cstring.h>`.

Input validation: don't worry about it for this part.

Side Note: Those of you who have looked closely at the signature of `strstr()` may notice that both of its arguments are `const`, meaning that you cannot modify the strings that they refer to. Originally, I was going to have the arguments of `strrstr()` be `const` as well, in order to match. However, because the return type is not `const`, I encountered some compiler errors that would have necessitated casting away the `const`-ness. Given this unnecessary detail and the fact that we haven't talked about `const`, I made the argument types not be `const` for `strrstr()`. I read more about `const` and `strstr()` here.

Here are examples of how your program should behave on a Linux command line.

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

#include "prog2.h"

int main()
{
    // One way to set up the call.
    char* ptr = strrstr("abcdabcb", "abc");
    printf("%s\n", ptr);

    // Could also set it up this way. (If strrstr() were to modify either string,
    // then I think that the above way would be prohibited because the literals
    // are likely placed in read-only memory.)
    char s1[] = "123ab67890123cd432123ef";
    ptr = strrstr(s1, "123");
    printf("%s\n", ptr);
    printf("%p %p\n", s1 + 18, ptr);
}
$ gcc -Wall -Werror test_strrstr.c prog2.c -o test_strrstr
$ ./test_strrstr
abcb
123ef
```

```
0x7ffda6678a62 0x7ffda6678a62
$
```

5.7 Part #5: `eachContains()`

In this part, you will implement `eachContains()`, which – as shown in `prog2.h` – has the following declaration:

```
int eachContains(char** strings, char target,
                 unsigned numStrings, char** firstOffending);
```

`strings` is an array of strings. The length of this array is given by `numStrings`. This function should return a true value (i.e. any nonzero value) if each string in the given array contains the given target character. In this case, nothing should be done with `firstOffending`; it won't be checked by the autograder. On the other hand, if at least one string *does not* contain the target character, then the function should return a false value (i.e. 0), and the `char` pointer referenced by `firstOffending` should be set to reference the first string in `strings` (i.e. the string at the lowest index in `strings`) that did not contain the target character.

Input validation: don't worry about it for this part.

Here are examples of how your program should behave.

```
$ cat test_eachContains.c
#include "prog2.h"

#include <stdio.h>

int main()
{
    char* strings[] = {"abcde", "hi there aaron", "apple", "banana"};
    char* firstOffending;
    if (eachContains(strings, 'a', 4, &firstOffending))
        printf("AAA\n");
    else
        printf("firstOffending: %s\n", firstOffending);
    if (eachContains(strings, 'b', 4, &firstOffending))
        printf("BBB\n");
    else
        printf("firstOffending: %s\n", firstOffending);
    if (eachContains(strings, 'e', 4, &firstOffending))
        printf("CCC\n");
    else
        printf("firstOffending: %s\n", firstOffending);
}
$ gcc -Wall -Werror test_eachContains.c prog2.c -o test_eachContains
$ ./test_eachContains
AAA
firstOffending: hi there aaron
firstOffending: banana
$
```