CS 211 Homework 1

Spring 2023

Code Due:	April 13, 2023, 11:59 PM, Central Time
Self-Eval Due:	April 16, 2023, 11:59 PM, Central Time
Partners:	No; must be completed by yourself

Purpose

The goal of this assignment is to get you programming with strings, iteration, and dynamic memory. Read through the requirements of the assignment carefully and then make sure to read the "Algorithm hints" section at the end of the document for additional guidelines on implementing the various requirements.

Preliminaries

Login to the server of your choice and *cd* to the directory where you keep your CS 211 work. Then unarchive the starter code, and change into the project directory:

```
% cd cs211
% tar -kxvf ~cs211/hw/hw1.tgz
⋮
% cd hw1
```

If you have correctly downloaded and configured everything then the project should build cleanly (although several of the tests may not pass because of the unimplemented code):

```
% make
⋮
cc -fsanitize=address,undefined -l211 -o test_translate...
%
```

Background

In this project, you will implement a clone of the standard Unix program *tr*(1), which is a *filter* program that performs transliteration. Given two equal-length sequences of characters, *from* and *to*, it replaces all occurrences of characters appearing in *from* with the character in the corresponding position in *to*.

The *tr* program takes the *from* and *to* character sequences as command-line arguments. Once the program is run, it takes in a string from user input, replaces the appropriate characters according to

This homework assignment must be completed on Linux by logging into a Linux workstation. Each time you login to work on CS 211, you should run *211* to ensure your environment is setup correctly. (If you get an error saying that `211.h` doesn't exist, that probably means you forgot to run *211*.)

Contents

the *from* and *to* sequences, and prints out the result. In the simplest case, the *from* and *to* sequences are strings of the same length. The following shows an example of your tr program could work.

```
% ./tr abc xyz
a
x
bbbcd
yyyzd
tag the cat
txg the zxt
abracadabra
xyrxzxdxyrx
^D
% echo Hello, world.  | ./tr e a
Hallo, world.
% echo Hello, world.  | ./tr elo 310
H3110, w0r1d.
% echo Hello, world.  | ./tr ',.  '  ___
Hello__world_
```

^D means press *Control-D*.

tr also understands ranges of characters and some backslash escape sequences:

```
% echo Hello, world.  | ./tr a-z A-Z
HELLO, WORLD.
% echo Hello, world.  | ./tr a-zA-Z. 'A-Za-z?'
hELLO, WORLD?
% echo Hello, world.  | ./tr a-zA-Z n-za-mN-ZA-M
Uryyb, jbeyq.
% echo Hello, world.  | ./tr a-zA-Z n-za-mN-ZA-M | ./tr
a-zA-Z n-za-mN-ZA-M
Hello, world.
% echo Hello, world.  | ./tr ' ' '\n'
Hello,
world.
%
```

When the | symbol is entered, it means that the output of the first command before the | symbol gets fed in as user input to the second command after the | symbol. In this first example, `echo Hello, world.` outputs "Hello, world." and so when the second command `./tr` is run, this output string is considered as the user input to be translated.

Characters that have special meaning for the shell, such as space, `!`, `*`, `?`, `$`, and `\`, need to be quoted in arguments.

The above examples won't work until you've finished the assignment, but **if you replace `./tr` with just `tr`, you should get the system's `/usr/bin/tr`, which will do the same thing.** Using the system's `tr` is a great tool for understanding the expected output in most cases, for example, if you are trying to write tests for expected behavior or when manually testing whether your program is working as expected.

Orientation

Your code is divided into three `.c` files:

- Most significant functionality will be defined in the "*translate* library," src/translate.c.
- Tests for those functions will be written in test/test_translate.c.
- The `main()` function that implements the *tr* program will be defined in src/tr.c.

Function signatures for src/translate.c are provided for you in src/translate.h; since the grading tests expect to interface with your code via this header file, **you must not modify src/translate.h in any way.** All of your code will be written in the three .c files.

Make *targets*

The project also provides a Makefile with several targets which you can enter after `make`:

target	description
test	builds everything & runs the tests * &
all	builds everything, runs nothing &
test_translate	builds the unit tests
tr	builds the *tr* program
clean	removes all build products &

* default & phony

Target test is the default, which means you can run it by typing `make` alone, with no target name.

Specifications

The project comprises two functional components, which are specified in this section. First, though, we define *charseqs* (character sequences).

Character sequences

The *tr* program uses charseqs to specify which characters to replace and what to replace them with. The C type of a charseq is just `char*`—that is, a C string—but they can be represented in two forms having different interpretations, each of which is used at different stages of the program:

- A *literal* charseq is just a sequence of characters, each standing for itself. For example, interpreted as a literal charseq, the string `"a-e"` contains the three characters `'a'`, `'-'`, and `'e'` at indices 0, 1, and 2, respectively. In a literal charseq, no character has special meaning.
- An *unexpanded* charseq may contain ranges, written "*c*-*d*", and escape sequences, written "*c*".

In C (but not C++) those literals don't actually have type `char`!—they have type `int` for obscure historical reasons. That is, `'A'` is an alternative way of writing the `int` value 65. Try printing `sizeof 'A'` and see. . . .

– The range "*c*-*d*" stands for the interval of characters from `'c'` to `'d'`, inclusive. (This means that if `'c' > 'd'` then the range is empty, and if `'c' == 'd'` then the range contains only `'c'`.)

  Range bounds, both lower and upper, are always represented by single characters. They are never the result of another range or escape expansion.

– If the escape "*c*" is valid C string literal escape sequence, then it has the same meaning for *tr* as in C; otherwise it just stands for character `'c'` itself.

– Every other character stands for itself. In particular, a "`-`" character that is not part of a range stands for itself, as does "\" character that is not followed by another character.

– In cases of ambiguity, the leftmost possible expansion takes priority, and a range takes priority over a potential escape at the same position.

We have provided you a function mapping character `'c'` to the meaning of *c*, so you don't have to figure that part out.

Here is a table showing several unexpanded charseqs along with their literal expansions, written as C string literals:

unexpanded		**literal**	
`"abc"`		`"abc"`	
`"a-e"`		`"abcde"`	
`"a-e_"`		`"abcde_"`	
`"a-df-i"`		`"abcdfghi"`	
`"-i"`		`"-i"`	
`"a-d-i"`		`"abcd-i"`	
`"\\t"`	(2 characters)	`"\t"`	(1 character)
`"\\-_"`	(3 characters)	`"\\]^_"`	(4 characters)
`"X-\\n"`	(4 characters)	`"XYZ[\\n"`	(6 characters)

How could we figure out what characters *should* appear in these ranges? See the manual page: `man ascii`.

The *tr* program takes charseqs in unexpanded form, and must expand them to literal form before it can do its work.

The translate *library*

The *translate* library is responsible for expanding charseqs from unexpanded to literal form, and for using a pair of literal charseqs to translate a string. It provides a function for each of these purposes that will be used in src/tr.c. Additionally, the header file (translate.h) exposes two helper functions which you will implement in translate.c to facilitate testing. See section "Algorithm hints" on page 9 for detailed hints on how to implement each of the functions you are required to write. src/translate.c defines four functions:

- Function `expand_charseq(const char*)` takes a charseq in unexpanded form and expands it, returning it in literal form.

  The returned charseq is allocated by *malloc*(3), which means that the caller (the function that called `expand_charseq`) is responsible for deallocating it with *free*(3) when finished with it. While you will need to implement the main expansion functionality of `expand_charseq`, we already handle the memory allocation for you and you will notice usages of *malloc*(3) and *free*(3) in the starter code already.

  Error case: If `expand_charseq()` is unable to allocate memory then it returns the special pointer value `NULL`. We implement this error case for you as well.

- Function `charseq_length(const char*)` is a helper to `expand_charseq()` that determines how long the literal result of expanding its argument will be.

- Function `translate(char* s, const char* from, const char* to)` takes a string to modify (`s`) and two literal charseqs (`from` and `to`). Each character in string `s` that appears in charseq `from` is replaced by the character at the same index in charseq `to`.

  To be precise: For each index `i` in `s`, if there is some `j` such that `s[i] == from[j]` (and there is no `k < j` such that `s[i] == from[k]`), then `s[i]` is replaced by `to[j]`.

  Undefined behavior: Function `translate()` has an *unchecked precondition*, whose violation will result in undefined behavior. In particular, for it to work properly, `from` must not be a longer string than `to`. However, `translate()` **should not** check this condition, as ensuring it is the caller's responsibility.

  A precondition of a function is a condition that needs to be true at the start of a function's execution for the function to execute safely (avoiding undefined behavior) and correctly

- Function `translate_char(char c, const char* from, const char* to)` is a helper to function `translate()`. It takes a character to translate (`c`) and two literal charseqs (`from` and `to`). It returns the translation of character `c` as given by the two charseqs.

  Undefined behavior: Function `translate_char()` has the same unchecked precondition as function `translate()`, with the same results if violated. (This is a natural consequence of `translate()` calling `translate_char()`.)

An additional unchecked precondition for all four of the above functions is that all `char*`s that they are given as arguments must be non-null pointers to `'\0'`-terminated character arrays—that is, valid C strings. If this precondition is violated then the functions' behaviors are undefined. (This means that these functions *should not* check whether their arguments are null.)

The tr *program*

The *tr* program must be run with two command-line arguments. The *tr* program has three phases: first it validates and interpets its arguments, then it transforms its input to its output, and then it cleans up its resources.

We've provided you with the first check, for the correct number of arguments passed to the program. This serves as an example of how to use *fprintf*(3) and *stderr*(4) for printing error messages.

Next, `expand_charseq()` is called to expand both command-line arguments `argv[1]` and `argv[2]` into literal charseqs. Since `expand_charseq()` returns `NULL` if it cannot allocate memory, you need to `NULL`-check both results; if it fails, print the error message as below (using `OOM_MESSAGE` and `argv[0]`), call `free()` on `from` and `to`, and exit with error code 10.

Two calls to `expand_charseq()` mean both resulting literal charseqs require two calls to `free()` in order to clean up their allocated memory at the end

```
tr: error: out of memory
```

If character sequence expansion succeeds but the charseqs, once expanded, don't have the same length, it is an error; your code should print the specified error message (`LENGTH_MESSAGE`) to `stderr` (as below), call `free()` on `from` and `to`, and exit with error code 2.

```
tr: error: lengths of FROM and TO differ
```

Now, if there are no errors then the program is ready to iterate over the input lines until `read_line()` returns `NULL`, translating each line and printing the result. This is also already implemented for you. Since each input line read by `read_line()` is allocated by `malloc()`, each line is freed with `free()` when it is done being used.

Now that argument checking has succeeded, *tr* begins taking in strings to translate. For each line read from the standard input, it translates the line according to the literal expansions of `FROM` and `TO` and prints the result. When there is no more input to process, the program terminates successfully.

The examples in the *Background* section involve sending your `tr` program one line at a time. Be sure to test it interactively, too, to make sure it handles multiple lines correctly:

```
% ./tr a-z A-Z
Be sure to test
BE SURE TO TEST
your program
YOUR PROGRAM
interactively.
INTERACTIVELY.
^D
%
```

Reference

Accepting command-line arguments

When running a C program from the command line, the user can supply it with *command-line arguments*, which the program's `main()` function then receives as an array of strings. In particular, `main()` can be declared to accept two function arguments, as follows:

```
int main(int argc, char* argv[]);
```

Then `argc` will contain the number of command-line arguments (including the name of the program itself in `argv[0]`), and `argv` will contain the comnand line arguments themselves.

For example, if a C program is run like

```
% my_prog foo bar bazzz
```

then `argc` is 4 and `argv` is the array

```
{
    "my_prog",
    "foo",
    "bar",
    "bazzz"
}.
```

Reading input a line at a time

The C programming language doesn't provide an easy way to read a line of input whose length is unknown, so we have provided you a small library, *lib211*, on the Unix login machines. The library exports a function `read_line()` for this purpose. Here is its signature:

```
char* read_line(void);
```

The `read_line` function returns a character array allocated by *malloc*(3), which means that the caller is responsible for deallocating it with *free*(3) when finished with it. See the next subsection for more on this topic, and see the *read_line*(3) manual page on the lab machines for information on the *read_line* function.

C natively provides *gets*(3), which is easy to use but inherently unsafe, and *fgets*(3), which can be used safely but requires you to specify a limit on the length of the line.

Working with C strings

When testing your functions, you might be tempted to write assertions like this:

```
assert( expand_charseq("a-e") == "abcde" );
```

But there are three problems with this:

1. It leaks memory.
2. It compares the addresses of the strings rather than the characters in them.
3. In rare cases, it might cause undefined behavior.

The second and third problems here are also solved by `CHECK_STRING`, which is described in the next subsection.

It leaks memory because `expand_charseq()` allocates memory and the code above doesn't free it. To fix that, we need to store the result of `expand_charseq()` in a variable, which lets us refer to it twice:

```
char* actual_result = expand_charseq("a-e");
assert( actual_result == "abcde" );
free(actual_result);
```

However, this still won't work, because when you use `==` to compare pointers, it compares *the addresses*, not the pointed-to values. And the address returned by `expand_charseq()` will never be the same as the address of a string literal.

Instead, to compare strings, we need to use the *strcmp*(3) function (from `<string.h>`), which compares them character by character. You may expect, incorrectly, that *strcmp*() would return `true` for equal strings and `false` for unequal strings, but actually it does something more useful: `strcmp(s1, s2)` determines the lexicographical ordering for `s1` and `s2`. If `s1` should come before `s2` when sorting then it returns a negative `int`; if `s1` should come after `s2` then it returns a positive `int`. If they are equal, it returns `0`. Thus we should write:

Lexicographical order is a generalization of alphabetical order to sequences of non-letters (or more than just letters). *strcmp*() compares the numeric values of `char`s, which means that `'a' < 'b'` and `'A' < 'B'`, but also `'B' < 'a'` and `'$' < ','`.

```
char* actual_result = expand_charseq("a-e");
assert( strcmp(actual_result, "abcde") == 0 );
free(actual_result);
```

This almost works! In fact, it usually will work. But to be completely correct, we need to deal with the possibility that `expand_charseq()` fails to allocate memory and returns `NULL`. In that case, *strcmp*() will deference `NULL`, which is undefined behavior. Thus, we need to ensure that `actual_result` is not `NULL` before we try to use the string that it points to:

```
char* actual_result = expand_charseq("a-e");
assert( actual_result );
assert( strcmp(actual_result, "abcde") == 0 );
free(actual_result);
```

Here are some more functions from `<string.h>` that you may find useful:

`char* strchr(const char* s, int c)`
: Searches string `s` for the first occurrence of `(char)c`, returning a pointer to the occurrence if found or `NULL` if not.

Why does *strchr*() take an `int` rather than a `char`? Many C functions take a character as type `int` for obscure historical reasons.

`char* strcpy(char* dst, const char* src)`
: Copies string pointed to by `src` into string pointed to by `dst` (which must have sufficient capacity, or you'll get undefined behavior).

`size_t strlen(const char*)`
: Computes the length of a string (not including the `'\0'`).

Better testing assertions

We have created a number of helpful functions for testing your code, available in the *lib211* library. They have names like `CHECK()` or `CHECK_INT()` and function similarly to assertions: either the statement is true or the test fails. Here's what writing test assertions with these macros looks like:

```
static void example_checks(void)
{
    CHECK_INT( 2 * 3, 6 );
    CHECK_SIZE( sizeof(double), 8 );
    CHECK_CHAR( toupper('a'), 'A' );
    CHECK( islower('a') );
}
```

The difference between `CHECK(a == b);` and `CHECK_INT(a, b);` is that the latter prints the values of `a` and `b` when it fails, whereas the former does not.

The provided checks are summarized here:

Form . . .	**checks that** . . .
`CHECK_CHAR(x, y);`	`x` and `y` are equal `char`s
`CHECK_INT(x, y);`	`x` and `y` are equal `int`s
`CHECK_UINT(x, y);`	`x` and `y` are equal `unsigned int`s
`CHECK_SIZE(x, y);`	`x` and `y` are equal `size_t`s
`CHECK_DOUBLE(x, y);`	`x` and `y` are equal `double`s
`CHECK_STRING(x, y);`	`x` and `y` point to equal `'\0'`-terminated strings
`CHECK_POINTER(x, y);`	`x` and `y` point to the same object
`CHECK(x);`	`x` is `true`, non-zero, or non-null

Algorithm hints

In this section, we provide suggestions, such as algorithms, for writing the necessary functions. These hints are given in what we expect will be the best order of implementation. It's a very good idea to test each function as you write it, rather than testing them all at the end, because you will find bugs sooner that way.

The `charseq_length()` function

The `charseq_length()` function scans its argument string (an unexpanded character sequence) while counting how many characters it will take when expanded. Thus, you need two variables: one to count, and one to keep track of the position while scanning the string. Start the count at 0 and the position at the beginning of the argument string (i.e., at `o`). Then iterate through the sequence string and evaluate the following conditions for each iteration:

To keep track of a position in a string, you can use a `size_t` variable to hold the index.

- If the character at the current position is `'\0'`, then you've reached the end and should return the count.
- If the character at the *next* position is `'-'`, and the character at the position after that is not `'\0'`, then you've found a range. If we call the character before the hyphen `start` and the character after the hyphen `end`, then we can determine the length of the range by comparing the two characters: If `start > end` then the range is empty; otherwise the length of the range is `end - start + 1`. Add this resulting length to the count, and then advance the current position by 3 to get to the first character past the right side of the range.

  This implies that a hyphen at the beginning or end of the string, or immediately following the end of a character range, is interpreted literally rather than denoting a range.

  Remember that `char`s are integers and so can be subtracted
- If the character at the current position is `'\\'` (a single backslash), and the character at the next position is not `'\0'` then you have found an escape sequence. Its expanded length is 1, so add that much to the count, and advance the current position by 2 to get to the first character after the escape sequence.

  This case should be checked after the range case, which implies that the literal expansion of unexpanded charseq "`\-_`" is "`\]^_`", not "`-_`".
- Otherwise, the character at the current position will be copied as is, so increment the count by 1 and advance the current position to the next position in the string.

The `expand_charseq()` function

Like `charseq_length()`, the `expand_charseq()` function scans its argument string (an unexpanded character sequence), but instead of counting, it copies the characters into a fresh string, expanding ranges and escape characters into their literal meanings.

The first thing it must do is allocate memory for its result. We have provided you code that calls `charseq_length()` to find out how much memory is needed, allocates the memory, and checks that the allocation succeeded. Then the algorithm works by scanning the argument string while storing characters into the result string. To do this, you will likely need two variables: one to keep track of your position in the unexpanded character sequence being scanned (the source); and one to keep track of your position in the result string being filled in (the destination).

This function is probably the trickiest part of the whole homework. One way to develop your code would be to hold off writing this function and move forward, while temporarily considering all input charseqs to be literal. It's not hard to add a call to `expand_charseq()` to `src/tr.c`'s `main()` function once you get it working.

The control logic of the scanning-and-copying loop is the same as in the `charseq_length()` function (including by iterating through the source sequence), but the actions at each step differ:

- If the character at the current source position is `'\0'`, then you've reached the end. Don't forget to store a `'\0'` at the destination position (which should be the end of the result string) before returning.

- If the character at the *next* source position is `'-'`, and the character at the position after that is not `'\0'`, then you've found a range. If we call the character before the hyphen `start` and the character after the hyphen `end`, then we can generate the range by iteration, incrementing `start` until it passes `end`. That is, while `start <= end`, we want to store `start` to the destination position, advance the destination position, and increment `start`. Once we've fully expanded the range, we advance the source position past it (by adding 3).

To avoid undefined behavior here, you should store `start` and `end` as `int`s, not `char`s. To understand why, consider what would happen if `end` were `CHAR_MAX`.

- If the character at the current source position is `'\\'`, and the character at the next source position is not `'\0'` then you have found an escape sequence. Its expansion is given by `interpret_escape(c)` (provided in `src/translate.c`), where `c` is the character following the backlash. Store the resulting expansion to the destination position, advance the destination position, and advance the source position past the escape sequence (by adding 2).
- Otherwise, the character at the current position stands for itself, so store it at the current destination position and then advance both the source and destination positions by 1.

The `translate_char()` function

The `translate_char()` function takes a character to translate (`c`) and two literal charseqs (`from` and `to`). The idea is to scan charseq `from` searching for `c`. If we find `c` at some index `i` then return `to[i]`. If we get to the end of `from` without finding `c` then return `c` unchanged.

The `translate()` function

The `translate()` function takes a string to translate in place (`s`) and two literal charseqs (`from` and `to`). The idea is to iterate through each character in `s`, replacing each character with its translation according to `translate_char()`.

Deliverables & evaluation

For this homework you must:

1. Implement the specification for the *translate* library from the previous section in src/translate.c.
2. Implement the specification for the *tr* program from the previous section in src/tr.c.
3. Add more test cases to test/test_translate.c in order to test the four functions that you defined in src/translate.c.

The file test/test_translate.c already contains two tests cases for each of the four functions, and helper functions to facilitate testing for two of them. Because the functions you are implementing are complex and have many corner cases, you need to add many more tests for each. Try to cover all the possibilities, because **you will be required to fill out a self-evaluation after you submit this assignment**, where we will spot-check your test coverage by asking for just a few particular test cases. You can't anticipate which we'll ask about, so you should try to cover everything.

Take a look at the existing tests to get an idea of what tests can look like. You can also write tests whose inputs will cause the various branches of a function (e.g., `if`/`else` statements, loops) to execute so that your tests cover as many possibilities of code behavior as possible.

Grading will be based on:

- the correctness of your implementations with respect to the specifications,
- the presence of sufficient test cases to ensure your code's correctness (tested via the self-evaluation), and
- **adherence to the CS 211 Style Manual (see manual here).**

Submission

Homework submission and grading will use Gradescope. You must include any files that you create or change. For this homework, that will include src/translate.c, src/tr.c, and test/test_translate.c. (You must not modify Makefile or src/translate.h.)

Per the syllabus, if you engaged in arms-length collaboration on this assignment, you must cite your sources. You may write citations either in comments on the relevant code, or in a file named README.txt that you submit along with your code. See the syllabus for definitions and other details.

Submit using the command-line tool `submit211`. You can run the command with the `--help` flag to see more details. The tool will ask you to log in with your Gradescope credentials, so make sure you've created an account!

To submit the necessary files for this homework, you will run something that looks like:

```
% submit211 submit --hw hw1 src/translate.c src/tr.c test/test_translate.c
```

Remember that those are relative paths to the files you want to submit. So make sure to change them to make sense for whatever

directory you are running the command from. You can also add any additional files you want to upload, like README.txt, to the end of the command.

For this assignment, you will have unlimited submissions and all tests are visible to you. In future assignments, you may have limited submissions and not all tests will be visible to you to ensure your code works on unseen cases.

Gradescope Results

Some characters in the autograder tests are non-printable characters. In those cases, we instead represented the character as a hexadecimal sequence. These can be found in the "Hex" column of the ASCII table. For example: `\x09` represents the ASCII value of `0x09`, which is the tab character. `\x0A` represents the ASCII value of `0x0A`, which is the new line character. `\x7F` represents the ASCII values of `0x7F`, which is the delete character. In each case, the hexadecimal sequence represents a single, non-printable character, and a warning is added the line of output, stating "(warning: translated to hexadecimal sequence)".