Purpose

The goal of this assignment is to get you programming with strings, iteration, and dynamic memory.

Preliminaries

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

```bash
$ curl $URL211/hw/hw0R.tar.gz | tar rvx
...$ cd hw0R
```

If you have correctly downloaded and configured everything then the project should build cleanly:

```bash
$ make all
...
cc -o build/test_translate build/test_translate.o b...
$ 
```

Background

In this project, you will implement a clone of the standard Unix utility `tr(1)`, which is a filter that performs transliteration. Given two equal-sized sets 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` sets as command-line arguments. In the simplest case, they are strings of the same length:

```bash
$ echo 'Hello, world!' | build/tr e a
Hallo, world!
$ echo 'Hello, world!' | build/tr elo 310
H3110, w0r1d!
$ echo 'Hello, world!' | build/tr ',! ' ___
Hello_ _world_
```

Filter programs copy their standard input to their standard output while modifying it in some way. For example `grep(1)` prints only lines that match some given pattern; `head(1)` discards all but the first `n` lines.

Characters that have special meaning for the shell, such as space, !, *, ?, $, and \, need to be quoted in arguments.
The shell command alias lets you define a shorter name for a longer command. (If your shell is not tcsh(1) then the syntax for alias may be different. In bash(1) and zsh(1), it would be alias rot13='...'.)

The above examples won’t work until you’ve finished the assignment, but if you replace build/tr with just tr, you should get the system’s /usr/bin/tr, which will do the same thing.

**Orientation**

As in Homework 1, 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.

The project also provides a Makefile with several targets:

<table>
<thead>
<tr>
<th>target</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>builds everything†‡</td>
</tr>
<tr>
<td>test</td>
<td>builds and runs the tests†</td>
</tr>
<tr>
<td>build/test_translate</td>
<td>builds (but doesn’t run) the tests</td>
</tr>
<tr>
<td>build/tr</td>
<td>builds the tr program</td>
</tr>
<tr>
<td>clean</td>
<td>removes all build products†</td>
</tr>
</tbody>
</table>

* default  † phony
Specifications

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

Character sets

The tr program uses charsets to specify which characters to replace and what to replace them with. The C type of a charset is just char—that is, a C string—but they can be represented in two forms having different interpretations:

• A literal charset is just a sequence of characters, each standing for itself. For example, interpreted as a literal charset, the string "a-e" contains the three characters 'a', '-', and 'e' at indices 0, 1, and 2, respectively. In a literal charset, no character has special meaning.

• An unexpanded charset may contain ranges, written “c-d”, and escape sequences, written “\c”.

  – 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'.)

  – 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.

Here is a table showing several unexpanded charsets along with their literal expansions:

<table>
<thead>
<tr>
<th>unexpanded</th>
<th>literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abc&quot;</td>
<td>&quot;abc&quot;</td>
</tr>
<tr>
<td>&quot;a-e&quot;</td>
<td>&quot;abcde&quot;</td>
</tr>
<tr>
<td>&quot;a-e_&quot;</td>
<td>&quot;abcde_&quot;</td>
</tr>
<tr>
<td>&quot;a-df-i&quot;</td>
<td>&quot;abcdefghi&quot;</td>
</tr>
<tr>
<td>&quot;\t&quot; (2 characters)</td>
<td>&quot;\t&quot; (1 character)</td>
</tr>
</tbody>
</table>

The tr program takes charsets 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 charsets from unexpanded to literal form, and for using a pair of literal charsets 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 exposes two
helper functions to facilitate testing. Thus, src/translate.c defines four functions:

- **Function expand_charset(const char*)** takes a charset in expanded form and expands it, returning it in literal form.

  The returned charset is allocated by malloc(3), which means that the caller is responsible for deallocating it with free(3) when finished with it.

  **Error case:** If expand_charset() is unable to allocate memory then it returns the special pointer value NULL.

- **Function charset_length(const char*)** is a helper to expand_charset() 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 charsets (from and to). Each character in string s that appears in charset from is replaced by the character at the same index in charset 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.

- **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 charsets (from and to). It returns the translation of character c as given by the two charsets.

  To be precise: If there is some j such that c == from[j] (and there is no k < j such that c == from[k]), then this function returns to[j]; but if there is no such j then it returns c unchanged.

  **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().)

*The tr program*

The tr program must be run with two command-line arguments. If run with more or fewer than two, it prints the message
Usage:  tr FROM TO < INPUT_FILE

to stderr, where tr is replaced by argv[0] (the actual name that the
program was called with), and then exits with error code 1.

The arguments FROM (argv[1]) and TO (argv[2]) are unexpanded
charsets, so tr must expand them to literal charsets. If the lengths of
the two literal charsets differ (post-expansion, that is) then it prints
the message

  tr: error: lengths of FROM and TO differ

to stderr, where again tr is replaced by argv[0], and then exits with
error code 2.

Now that argument checking has succeeded, tr begins filtering.
For each line read from the standard input, it translates the line ac-
cording to the literal expansions of FROM and TO and prints the result.
When there is no more input to process, the program terminates
successfully.

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 command 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 I have provided you a small library, lib211, with your Homework 2 starter code. The lib211.h header declares a function read_line() that reads a line from the standard input and returns it.

The 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.

Managing memory with malloc(3) and free(3)

In Homework 1, all memory used by your program was allocated and deallocated automatically. But to work with strings, especially strings whose length is not known when the program is written, we need a different technique.

Function malloc() (from <stdlib.h>) takes the number of bytes that you need and attempts to allocate that much memory. For example, we can allocate enough memory for one int, or for an array of N ints:

```c
int* just_one = malloc(sizeof(int));
int* several = malloc(N * sizeof(int));
```

If malloc() succeeds, it returns a pointer to the newly allocated memory, which can be used to hold any type that fits. The memory this pointer points to is uninitialized, so you must initialize it to avoid undefined behavior. When you are done with this memory, you must free it by passing the pointer to free().

If malloc() fails to find sufficient memory, which it can, it returns the special pointer value NULL, which is a valid pointer that points nowhere. Dereferencing NULL is undefined behavior, but you can compare it using the == operator. Consequently, every call to malloc() must be followed by a NULL check. We provide this call to malloc() and the obligatory NULL check in src/translate.c:

```c
char* result = malloc(charset_length(src) + 1);
char* dst = result;

if (result == NULL) return NULL;
```

Two things to note about the above malloc() call:

- We are allocating one more byte than the length that src will expand to, because we need an extra byte to store the string’s ‘\0’ terminator.

It 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.
• There is no need to multiply the desired number of **chars** by `sizeof(char)` because `sizeof(char)` is always 1.

**Working with C strings**

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

```c
assert ( expand_charset("a-e") == "abcde" );
```

But there are three problems with this.

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

```c
char* actual_result = expand_charset("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_charset()` will never be the same as the address of a string literal.

Instead, to compare strings, we need to use the `strcmp()` function (from `<string.h>`), which compares them character by character. You may expect 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:

```c
char* actual_result = expand_charset("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_charset()` 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:

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

**Lexicographical order** is a generalization of alphabetical order to sequences of non-letters (or more than just letters). `strcmp()` compares the numeric values of **chars**, which means that ‘a’ < ‘b’ and ‘A’ < ‘B’, but also ‘B’ < ‘a’ and ‘$’ < ‘,’.
Here are some more functions from `<string.h>` that you may find useful:

```c
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.

```c
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 `UB`).

```c
size_t strlen(const char*)
```
computes the length of a string (not including the `\0`).

### 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.

### Algorithm for the `charset_length()` function

The `charset_length()` function scans its argument string (an unexpanded character set) 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. Then iterate and evaluate the following conditions for each iteration:

- 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 to the count, and then advance the current position by 3 to get to the first character past the right side of the range.
- 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

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

To scan a string you can use either an index `size_t i` or pointer `char* p`. If you hold onto the original string `s` then the two approaches are interchangeable, since `p == s + i`, or equivalently `i == p - s`.

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.

This case should be checked after the range case, which implies that the literal expansion of unexpanded charset “\-_” is “\^_”, not “\_.”
much to the count, and advance the current position by 2 to get to the first character after the escape sequence.

- 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 character.

**Algorithm for the expand_charset() function**

Like charset_length(), the expand_charset() function scans its argument string (an unexpanded character set), 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 charset_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 three variables: one to remember the start of the result string in order to return it; one to keep track of your position in the unexpanded character set being scanned (the source); and one to keep track of your position in the result string being filled in (the destination).

The control logic of the scanning-and-copying loop is the same as in the charset_length() function, 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, so long as \( \text{start} \leq \text{end} \), 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).

- 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

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 charsets to be literal. It’s not hard to add a call to expand_charset() to src/tr.c’s main() function once you get it working.

To avoid undefined behavior here, you should store start and end as `ints`, not `chars`. To understand why, consider what would happen if end were `CHAR_MAX`.
the backlash. Store the 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 traditional C way to do this is *dst++ = *src++;

Algorithm for the translate_char() function

The translate_char() function takes a character to translate (c) and two literal charsets (from and to). The idea is to scan charset 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.

Algorithm for the translate() function

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

Algorithm for the tr program

The tr program has three phases: first it validates and interprets 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. This serves as an example of how to use fprintf(3) and stderr(4) for printing error messages.

Next, use expand_charset() to expand both command-line arguments argv[1] and argv[2] into literal charsets. Since expand_charset() returns NULL if it cannot allocate memory, you need to NULL-check both results; if it fails, print the error message (using OOM_MESSAGE and argv[0]) and exit with error code 2.

If character set expansion succeeds but the charsets, once expanded, don’t have the same length, it is an error; print the specified error message (LENGTH_MESSAGE) to stderr and exit with error code 2.

Now, if there are no errors then we are ready to iterate over the input lines until read_line() returns NULL, translating each line and printing the result. Since each input line read by read_line() is allocated by malloc(), you need to free each line with free() when you are done with it. This should be straightforward because you process one line at a time and never need to hold onto one longer.

Two calls to expand_charset() mean you will need two calls to free() in order to clean up in the end.
Deliverables and 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_convert.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 for this week’s self evaluation 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.

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, and

• adherance to the EECS 211 Style Manual.

Submission

Homework submission and grading will use the GSC grading server. You must upload 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 should not need to modify Makefile and you must not modify src/translate.h.)

Submit using the command-line GSC client gsc(1). Instructions are available in the submit211(7) manual page on the lab machines. To view it, run:

    $ man submit211