CS351 Lab #2

Utilities

Instructions:

- *Assigned date: Tuesday February 1st, 2022*
- *Due date: 11:59PM on Friday February 11th, 2022*
- *Maximum Points: 32*
- *This lab must be done individually*
- *Please post your questions to BB*
- *Only a softcopy submission is required; it will automatically be collected through GIT at the deadline; email confirmation will be sent to your HAWK email address; late submission will be penalized at 10% per day; your submission will not be graded until you have submitted a single page PDF file with your name, email, A#, assignment number, and URL to the gitlab repo*

1 Your Assignment

Objectives

1. Practice programming in C
2. Practice using the Unix command line
3. Practice using provided build and testing tools
4. Learn how to consult the Unix manual pages for library documentation
5. Use C standard library functions for I/O

Overview

For this machine problem you'll hone your C programming skills and familiarize yourself with some standard library functions (for I/O, in particular) by implementing your own versions of three standard Unix utilities: tr, zip, and unzip. Along the way we also hope you'll grow more comfortable with the command line.

We describe the three utilities (as you'll implement them) next.

tr

The tr ("translate") utility, per the manual page, "copies the standard input to the standard output with substitution or deletion of selected characters." It is convenient in situations where we'd like to convert between line ending characters, lower/uppercase text, delete extraneous characters, etc.

When invoking `tr`, we can provide it with two strings of equal length. The first string is the list of characters to replace, and the second is the list of characters to replace them with.

Here's a typical interaction --- notice that because `tr` uses standard input and the command line buffers input by line, after invoking the utility it translates input on a line-by-line basis. (The line starting with '`$`' is the command prompt and entered command; this is followed by alternating lines of input and output text.)

```
$ tr abc 123
abracadabra
12r131d12r1
A man a plan a canal
A m1n 1 pl1n 1 31n1l
```

To end input we use the `^D` (Ctrl-D) keypress, which sends an end-of-file (EOF) character to `tr`.

Here's another interaction where we use the `-d` flag to indicate that we want to delete the characters in the string from the input.

```
$ tr -d abc
abracadabra
rdr
a man a plan a canal
 mn  pln  nl
```

When we want to use `tr` to process the contents of a file, we typically do so using a shell feature known as *I/O redirection*. Suppose we have a file named "test.txt" with the following data:

```
apples,bananas,cats
this is not a fruit
```

We can run `tr` on it as follows:

```
$ tr ', ' ' -' < test.txt
apples bananas cats
this-is-not-a-fruit
```

The '<' character indicates that the shell should take the contents of the named file ("test.txt") and use it as standard input to tr. Also note that the single quotes used around the strings at the command line allow us to include spaces (and other special characters) in the replacement/substitution strings --- the quotes themselves are not sent as part of the command line arguments to the program.

Below we use tr on the same file, with the -d option:

```
tr -d ', ' < test.txt
applesbananascats
thisisnotafruit
```

zip

zip is a compression utility. The actual Unix zip utility supports a number of different compression algorithms, but we'll be using a very simple form of compression known as run-length encoding (described below). zip will take a filename when invoked and output the compressed version of that file to standard output.

Because the output of zip is not intended to be human readable, we use I/O redirection again to send the compressed output to a file. Here's how we might use zip to compress the contents of the file "test.txt" into "test.zip" (note that the version of zip on Fourier has a different syntax: “$ zip test.zip test.txt”).

```
$ zip test.txt > test.zip
```

The '>', in this case, tells the shell to send the standard output of zip into the named file on the right.

The run-length compression algorithm works by simply scanning for identical adjacent bytes in the input file and printing just a single copy to the output preceded by a count. For instance, if the input is as follows:

```
aaaaaaaaaaaaaaaaaaaabbbbbbbbbbccccdddde
```

Run-length encoding would nominally output:

```
20a10b5c3d1e
```

Critically, however, since we need to be able to read and decode the compressed output (say, to obtain the original uncompressed version), the encoder will consistently print out each count as a 4-byte integer. This means that while the input to zip may be ASCII (and therefore human-readable), its output will not be. We can use another Unix utility -- od ("octal dump") -- to view the contents of non-human-readable (aka. binary) files. Assuming the sample input above (aaaaaaaaaaaaaaaaaaabbbbbbbbbbccccccddde) is saved in a file named "test.txt", here's a sample interaction.

```
$ od -t x1z test.txt
0000000 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61  >aaaaaaaaaaaaaaaa<
0000020 61 61 61 61 62 62 62 62 62 62 62 62 62 63 63  >aaaabbbbbbbbbbcc<
0000040 63 63 63 64 64 64 65 0a                    >cccddde.<
0000050

$ zip test.txt  > test.zip

$ od -t x1z test.zip
0000000 14 00 00 00 61 0a 00 00 00 62 05 00 00 00 63 03  >....a....b....c.<
0000020 00 00 00 64 01 00 00 00 65 01 00 00 00 0a        >...d....e.....<
0000036
```

We start by viewing the contents of "test.txt" using od (read the manual page for od for an explanation of the flags we use). This tells us that the ASCII codes for a, b, c, ... are 61, 62, 63, ... We also see 0a at the end of the file, which is the newline character.

After zip-ping the file, od shows us that the run-length encoded version consists of 30 total bytes. Each 5-byte sequence consists of a 4-byte integer (encoded in little-endian) followed by a 1-byte ASCII code from the uncompressed file. All values are shown in hex (e.g., 0x14 is decimal 20).

Because of the 4-byte integer encoding, the maximum count value that can be written is 4,294,967,296. While this is theoretically a problem, you don't need to worry about it for the assignment (it can also be easily solved by separating over-long runs of identical bytes into separate run-length encodings).

unzip

unzip is invoked with the filename of a file compressed by zip, and prints out the uncompressed version to standard output.

Given the output file "test.zip" from the previous example, here's unzip in action:

```
$ unzip test.zip
aaaaaaaaaaaaaaaaaaaabbbbbbbbbbcccccddde
```

Implementation Details

Your implementations of tr, zip, and unzip, will go into the mytr.c, myzip.c, and myunzip.c files, found in your lab repository. **Note that you should only change the mytr.c, myzip.c, and myunzip.c files. You should not create any additional source files or external dependencies, as our script will not copy those for grading purposes (and your program will fail to build/run).**

The working specifications of the three utilities are presented in the previous section, but there are some details / edge cases to consider:

1. When the commands are invoked without any arguments or the incorrect number of arguments, they should print usage information and exit with error code 1. The correct usage output is already included in the provided skeleton code.
2. If tr is given replacement and substitution strings of different lengths, it should print the error message "STRING1 and STRING2 must have the same length" and exit with error code 1.
3. If the specified file doesn't exist (or can't be opened for another reason), both zip and unzip should print an error and exit with error code 1.
4. When the utilities are invoked with valid arguments and run to completion, they should terminate with exit code 0.

I/O and String library functions

A number of standard library functions should prove helpful in your implementation. First, a list of them (below their required header files) for easy reference:

```
#include <stdio.h>

int    printf(char *format, ...);

FILE  *fopen(char *path, char *mode);
int    fclose(FILE *stream);
```

```
int    fgetc(FILE *stream);
int    fputc(int c, FILE *stream);

size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
size_t fwrite(void *ptr, size_t size, size_t nitems, FILE *stream);


#include <string.h>

int    strcmp(char *s1, char *s2);
size_t strlen(char *s);
```

To look up the manual page for a function, use the command "man 3 FUNC_NAME". The 3 refers to section 3 of the manual pages, dedicated to library functions. (Section 2 is for system calls, which will come in handy later in the semester.) We'll give you a brief overview of the functions, but you have plenty of manual-page reading ahead of you --- best get started soon!

- printf: formatted printing to standard output
- fopen and fclose: opening and closing files for reading/writing, depending on the specified mode. fopen returns a "stream" pointer, which can be passed to fclose. E.g., to read from a file:

```
FILE *fp = fopen("foo.txt", "r"); // open for reading
...                               // read from fp
fclose(fp);                       // close file stream
```

  Note that all processes start with three streams already initialized and ready for use: stdin, stdout, stderr (corresponding to standard input, output, and error).

- fgetc and fputc: read/write a single character at a time from/to a stream. While they return/take the int as an argument, this is just to allow the special value EOF to be returned from fgetc when the end-of-file has been reached. Otherwise, the int values can be safely cast to unsigned chars. E.g., to print the contents of a file to standard output, one character at a time:

```
FILE *fp = fopen("foo.txt", "r");
while(1) {
  int c = fgetc(fp);
  if (c == EOF)
    break;
  fputc(c, stdout);
```

```
}
fclose(fp);
```

- fread and fwrite: read/write binary data from/to a stream. Each function takes a char array (for the data), the size of each piece of binary data to read/write (e.g., 4 bytes per int), the quantity of data (e.g., 10 ints), and a stream. E.g., this code reads 2 int sized numbers from the input file at a time, adds them, then writes the int-sized sum to standard output:

```
FILE *fp = fopen("bin.dat", "r");
int buf[2];
int sum;
while(1) {
  int nread = fread(buf, sizeof(int), 2, fp);
  if (nread < 2)
    break;
  sum = buf[0] + buf[1];
  fwrite(&sum, sizeof(int), 1, stdout);
}
fclose(fp);
```

- strcmp: returns 0 if the argument strings are identical; non-zero otherwise.
- strlen: returns the length of the null-terminated argument string (excluding the terminating null character).

Testing and Evaluation

Build the executables using the default Makefile target --- i.e., by just typing "make". This will generate the mytr, myzip, and myunzipfiles. You can run them manually with the commands ./mytr, ./myzip, and ./myunzip (the "./" means to look in the current directory for the named executable).

A test script is provided that runs 16 different tests defined in the "tests/" subdirectory. Each test is defined by at least five files, where the filename is the numerical identifier for the test, and the extension is one of desc, run, out, rc, err --- the contents of these files are described below:

- desc: briefly describes the purpose of the test
- run: the command used to run the test
- out: the correct standard output of the program
- rc: the expected exit code of the program after the test completes
- err: the correct standard error of the program (empty for all tests)

To run the test suite, simply use the target "make test". The first 6 tests are for mytr, and the next 10 are divided evenly between myzip and myunzip. If they all succeed, you'll see:

```
test 1: passed
test 2: passed
test 3: passed
test 4: passed
test 5: passed
test 6: passed
test 7: passed
test 8: passed
test 9: passed
test 10: passed
test 11: passed
test 12: passed
test 13: passed
test 14: passed
test 15: passed
test 16: passed
```

If a test fails, it will stop testing at that point and print out a brief explanation of the error (and how to go about locating the correct result).

Each test is worth 2 points; the machine problem has a maximum score of 32 points.

2 What you will submit

To submit your work, simply commit all your changes to the mytr.c, myzip.c, myunzip.c files and push your work to Github. You can find a git cheat sheet here: https://www.git-tower.com/blog/git-cheat-sheet/

Your solution will be collected automatically at the deadline. If you want to submit your homework later, you will have to push your final version to your GIT repository and you will have let the TA know of it through email. There is no need to submit anything on BB for this assignment. If you cannot access your repository contact the TAs.

Grades for late programs will be lowered 10% per day late.