CSC 209 Assignment 2, Summer 2022

Due by the end of Monday July 11, 2022; no late assignments without written explanation.

1. An encryption program

In labs we've seen the "rotation cipher", where we encrypt text by moving each letter thirteen spaces down in the alphabet, with wraparound. In those programs we hard-coded the number 13, but a better program would have this as a parameter — Julius Caesar's army used the rotation cipher with a rotation of 3 (so unlike rot13, it was not its own inverse; the inverse would be rot23).

In this assignment you will write a C program to implement a slightly more sophisticated cipher, where the parameter (and thus the decryption key) is not just a number from 0 to 25 but a word.

First of all, please note that the rotation code which you probably expressed like this (for the lower-case letter version) (or see my posted solutions in /u/csc209h/summer/pub/lab/soln/04/rot13.c or /u/csc209h/summer/pub/lab/soln/06/rot13.c):

```
        if (c >= 'a' && c <= 'm')
            putchar(c + 13);
        else if (c >= 'n' && c <= 'z')
            putchar(c - 13);
```

can be generalized using modular arithmetic to work with an arbitrary rotation factor *n* like so:

```
        putchar((c - 'a' + n) % 26 + 'a');
```

That is to say, first we subtract 'a' to convert the letter to a number from 0 to 25 indicating its position in the alphabet; then we add n (mod 26); then we add back the 'a' to convert it back to a letter. Easiest is to have separate versions for "if (islower(c))" and "if (isupper(c))".

Your assignment here is to write a cyclical rotation encrypter and decrypter which uses a word as a key as follows. This is sometimes known as a Vigenère cipher.

First of all, either you will have to interpret the encryption key in a case-insensitive way, or the encryption key must be all lower-case (in which case your program gives an error message if it contains any non-lower-case-letter characters) (in either case you give an error message if the encryption key contains any nonalphabetic characters).

Each character in the encryption key is converted to a number from 0 to 25, where 'a' is 0, etc.

The first character of the encryption key tells you the rotation amount for the first letter (whether upper- or lower-case) of the text to be encrypted. The second character of the encryption key tells you the rotation amount for the second letter. And so on, until you reach the end of the encryption key, at which point you start over. So if the encryption key is five letters, then a given letter in it indicates the rotation amount for every fifth letter in the text to be encrypted.

As before, non-letters are copied to the output unmodified (and they don't change where we are in the encryption key).

Your program will use getopt() to parse its command-line. There are two options. One is '−d', which means to decrypt instead of encrypting. The other option is '−k', which takes an argument which is the encryption key. After the options, your program takes zero or more filename arguments in the usual way — zero filename arguments means to read the standard input. Otherwise it processes all of the specified files, and it does **not** reset the encryption between files (e.g. if the encryption key is five letters long, and the first file has 124 letters in it, then the first letter of the second file will be encrypted based on the last letter of the encryption key).

You can either make −k mandatory (i.e. if there is no −k, your program gives a usage error), or you can have a default encryption key in its absence. Your usage message must describe the actual requirements of your program.

And do **NOT** store the text from the input files. Like the rot13.c program, the files should be processed character by character — read one character with getc() and output its transformed version with putchar(); no need for storage.

Please compare (with diff) your program's output to that of my solution in /u/csc209h/summer/pub/a2/crypt (compiled only, so that you don't see my solution prior to the due date!).

2. What year was Sunday, June 6th?

In this question you will write "whatyear.c", which answers questions such as "what year was Sunday, June 6th?"

Your program takes three command-line arguments: a weekday, a month, and a day. For the weekday and month, the user can use either a number or the name. The program checks whether that month and day was (or will be) that weekday this year; if not, it checks last year; and so on until it gets a match.

It will need to call time() and localtime() to find the current year; then it will call mktime() to generate a "struct tm" for that date in the applicable years so as to examine the tm_wday member to see whether or not that date was that weekday.

Most of the command-line parsing is supplied for you in /u/csc209h/summer/pub/a2/whatyear-starter.c. You need to write findord() which finds the weekday or month name in the list (the user of your tool can supply either weekday/month names or numbers). Your findord() needs to use strcasecmp() rather than strcmp() so as to do a case-insensitive search, but it doesn't need to be clever about partial matches like my supplied solution.

To find the weekday of a given date, call mktime(). You declare a struct tm (not just a pointer to one; you need the actual object), and you fill it up, and then if mktime() returns a non-negative result then the tm_wday member of your struct tm has been set appropriately. To call mktime(), the fields you need to set are: tm_year, tm_mon, tm_mday, tm_hour, tm_min, tm_sec, and tm_isdst. Use −1 for tm_isdst, which means that the library should use the value which was in effect at that time. For tm_hour, choose an hour in the middle of the day (rather than for example 0 for midnight, because that would run the risk of DST changing the date).

For tm_mon, note a looming off-by-one issue: If the user specifies a month number on the command line, it is from 1 to 12, but everything else expects the month number to be from 0 to 11. This is why there is a "− 1" after the atoi(argv[2]) call in whatyear-starter.c.
Even more bizarrely, the tm_year value is off by 1900, e.g. 2022 is represented as 122. This won't matter until your final output, at which point you can simply add 1900, as in the lab06 example.

Remember that you can use *cal* to check the answers.

3. find empty directories

Write "findempty.c", which finds empty directories in directory subtrees. It is run with one or more command-line arguments. Each is a directory to search, recursively. Output path names of all empty directories you find (one per line).

Note that path names should be reported in terms of the command-line arguments. For example, if there is an empty directory named /u/ajr/csc209/example/a2/something, then if I am cd'd to /u/ajr/csc209 and I run "findempty example" or "findempty example/a2", it would output "example/a2/something", whereas if I ran "findempty /u/ajr/csc209h/example", it would output "/u/ajr/csc209h/example/a2/something". That is to say, don't do anything too clever about path names! — just take the string offered by the user.

findempty should *not* do a chdir() (which you might be tempted to do to avoid string processing). If it does a chdir(), your program will no longer always work with multiple command-line arguments, since a chdir() in processing the first directory may invalidate the name of the specified second directory.

You may assume a maximum likely complete path name of, say, 2000 characters, so long as your program aborts with an error message, rather than exceeding array bounds, should the situation turn out to be more complex than this.

You may not use ftw() or fts().

Other notes

Your programs should behave like standard unix software tools. Error messages should be in standard formats, and to stderr rather than to stdout; the exit status of your program should be appropriate; the "crypt" program should take zero or more command-line file name arguments in the usual way; and you must use getopt() to parse the options where applicable (which is in crypt).

Please try sample implementations in /u/csc209h/summer/pub/a2 (compiled programs only, so that you can't see my sample solution source code before the due date!).

All of your programs must be in standard C. They must compile on the teach.cs computers with "gcc –Wall" with no errors or warning messages, and may not use linux-specific or GNU-specific features.

Pay attention to process exit statuses. Your programs must return exit status 0 or 1 as appropriate.

Call your assignment files crypt.c, whatyear.c, and findempty.c. Auxiliary files are not permitted. Submit with commands such as

```
        submit -c csc209h -a a2 crypt.c whatyear.c findempty.c
```

and the other 'submit' commands are as in assignment one and the labs.

Please see the assignment Q&A web page at https://www.teach.cs.toronto.edu/~ajr/209/a2/qna.html for other reminders, and answers to common questions.

Remember:

This assignment is due at the end of Monday, July 11, by midnight. Late assignments are not ordinarily accepted, and *always* require a written explanation. If you are not finished your assignment by the submission deadline, you should just submit what you have, for partial marks.

Despite the above, I'd like to be clear that if there *is* a legitimate reason for lateness, please do submit your assignment late and send me that written explanation.

And above all: Do not commit an academic offence. Your work must be your own. Do not look at other students' assignments, and do not show your assignment (complete or partial) to other students. Collaborate with other students only on material which is not being submitted for course credit.
