Choose one topic you are interested in. And write programs for it.

Topic 1: (Q7, page 585)

Of the many techniques for compressing the contents of a file, one of the simplest and fastest is known as run-length encoding. This technique compresses a file by replacing sequences of identical bytes by a pair of byte: a repetition count followed by a byte to be repeated. For example, suppose that the file to be compressed begins with the following sequence of bytes (shown in hexadecimal):

```
46 6F 6F 20 62 61 72 21 21 21 20 20 20 20 20
```

The compressed file will contain the following bytes:

```
01 46 02 6F 01 20 01 62 01 61 01 72 03 21 05 20
```

(e.g. 01 means 46 repeats once. 02 means 6F repeats twice.)

Run-length encoding works well if the original file contains many long sequences of identical bytes. In the worst case (a file with no repeated bytes), run-length encoding can actually double the length of the file.

(a) Write a program named `compress_file` that uses run-length encoding to compress a file. To run `compress_file`, we’d use a command of the form

```
compress_file  original_file
```

`compress_file` will write the compressed version of `original-file` to `original-file.rle`. For example, the command

```
compress_file  foo.txt
```

will cause `compress_file` to write a compressed version of `foo.txt` to a file named `foo.txt.rle`.

(b) Write a program named `uncompress_file` that reverses the compression performed by the `compress_file` program. The `uncompress_file` command will have the form

```
Uncompress_file  compressed_file
```

`Compressed_file` should have the extension .rle. For example, the command

```
Uncompress_file  foo.txt.rle
```

will cause `uncompress_file` to open the file `foo.txt.rle` and write an uncompressed version of its contents to `foo.txt`. `uncompress_file` should display an error message if its command-line argument doesn’t end with the .rle extension.

Note: submit two files `compress_file.c` and `uncompress_file.c`. 
One of the oldest known encryption techniques is the Caesar cipher, attributed to Julius Caesar. It involves replacing each letter in a message with another letter that is a fixed number of positions later in the alphabet. (If the replacement would go past the letter X, the cipher “wrap around” to the begging of the alphabet. For example, if each letter is replaced by the letter two positions after it, then Y would be replaced by A, and Z would be replaced by B.)

(a) Write a program that encrypts a message using a Caesar cipher. The user will enter the message to be encrypted and the shift amount (the number of positions by which letters should be shifted):

Enter message to be encrypted: Go ahead, make my day.
Enter shift amount (1-25): 3
Encrypted message: Jr dkhdg, pdnh pb gdb.

Notice that the program can decrypt a message if the user enters 26 minus the original key:

Enter message to be encrypted: Jr dkhdg, pdnh pb gdb.
Enter shift amount (1-25): 23
Encrypted message: Go ahead, make my day.

You may assume that the message does not exceed 80 characters. Characters other than letters should be left unchanged. Lower-case letters remain lower-case when encrypted, and upper-case letters remain upper-case. Hint: To handle the wrap-around problem, use the expression \(((\text{ch} - 'A') + n)\%26 + 'A'\) to calculate the encrypted version of an upper-case letter, where \text{ch} stores the letter and \text{n} stores the shift amount. (You’ll need a similar expression for lower-case letters.)

(b) Modifying above program so that the program prompts the user to enter the name of a file containing the message to be encrypted:

Enter name of file to be encrypted: message.txt.
Enter shift amount (1-25): 3

The program then write the encrypted message to a file with the same name but an added extension of .enc. In this example, the original file name is message.txt, so the encrypted message will be stored in a file named message.txt.enc. There’s no limit on the size of the file to be encrypted or on the length of each lien in the file.

Note: submit two files encrypt.c and encrypt_file.c.
Topic 3: (Q17, page 588)
Write a program that reads a series of phone numbers from a file and displays them in a standard format. Each line of the file will contain a single phone number, but the numbers may be in a variety of formats. You may assume that each line contains 10 digits, possibly mixed with other characters (which should be ignored). For example, suppose that the file contains the following lines:

404.817.6900
(215) 686-1776
312-746-6000
877 275 5273
61734342000

The output of the program should have the following appearance:

(404) 817-6900
(215) 686-1776
(312) 746-6000
(877) 275-5273
(617) 343-4200

Have the program obtain the file name from the command line.

Note: submit one file parsePhoneNumber.c.

All the topics are from your textbook “C programming: a modern approach” by K.N.King.

Submission:

• Upload both the source code and screenshots of your results as the topic requires to the folder named “Project2” of the dropbox in the desire2learn system. And please write your name as a comment at the first line in the file.