ECE-C201 Programming for Engineers
Programming Assignment 2

Run-Length Encoding & Decoding

1 Background

This section provides technical background you will need to complete this project.

1.1 Run-Length Encoding

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 attempts to compresses a file by replacing sequences of identical bytes with a pair of bytes: a **count byte** followed by a **data byte**. The count byte simply keeps track of the number of times the data byte was repeated before a different byte was encountered.

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 encoded file will contain the following bytes:

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

where the **bold** bytes are the **count bytes** and the underlined bytes are the data bytes.

Run-length encoding works well for compression 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.

1.2 Long Repeated Sequences

It is worth noting that since the **count byte** in this run-length encoding scheme is only 1 byte, it can only count up to `0xFF` (255) repeated **data bytes** in the file to be compressed.

However, your program ***must*** be able to encode any given file. A solution to this problem is to express run sequences exceeding 255 as multiple encoded byte pairs.

For example, if an input file has a sequence of 300 repeating instances of the byte `0x20`, this could be encoded as:

... **FF** 20 **2D** 20 ...

Notice how this will not require any corresponding modification when writing a decoder for run-length encoded files.

1.3 Magic Number

A Magic Number is a predetermined byte sequence located at the start of a binary file that marks it as being a particular file format. A Magic Number is generally between 2 to 6 bytes long, but could be longer. For example, all GIF89a image files start with the 6 byte sequence `47 49 46 38 39 61`, all MKV video files start with `1A 45 DF A3`, and UNIX scripts start with `23 21`, which is commonly known as the sha-bang due to its ASCII representation `#!` (ha**sh bang**!). Programs that read these types of files always first check that the Magic Number is present to ensure they are reading the correct type of file. In this project you will add the Magic Number `21 52 4C 45` to your run-length encoded files when you write them and check for it when you read them before reading the rest of the file.

2 The Supplied Code

I have provided to you an incomplete program called `rle`, which will compile and run as provided. The program has a functioning command line argument interface that allows the user instruct `rle` to perform one of four different actions. Each of these four modes of operation will be described in the following sections.

2.1 Compiling the Code

1. Create an account at `https://repl.it/signup` (it's free)
2. Make sure you are logged into your repl.it account
3. Visit `https://repl.it/@jshack/Project2` to get the project workspace. As soon as you begin editing main.c, the project will automatically fork into your account.
4. Type `gcc -o rle main.c` into the terminal on the right of your repl.it workspace. This will compile the program.
5. Run the `rle` program by typing `./rle` into the terminal.

As you can see, compiling and running the program works just like it would on an ordinary UNIX/Linux system.

Once you have run the program (without any command line arguments), `rle` will present you with the following usage information:

```
Usage: ./rle MODE filename

Available Modes:
  -c     Compress: Performs RLE compression on "filename"
                   and writes result to "filename.rle"

  -x     Expand: Performs RLE expansion on "filename". The
                 supplied "filename" must have the extension
                 ".rle" The result is written to "filename"
                 with the extension ".rle" removed.

  -d     Debug: Prints a hexdump of "filename" to the screen.

  -g     Generate: Writes the test file described in the Project 2
                   assignment document to disk as "filename". Use
                   this file to test and debug your program.

Examples:
  ./rle -c test.bin
        Produces RLE encoded file test.bin.rle
  ./rle -x test.bin.rle
        Expands test.bin.rle to disk as test.bin
  ./rle -d test.bin.rle
        Displays raw contents of test.bin.rle
  ./rle -g test.bin
        Generates test file with known contents to disk as test.bin
```

As you can see, `rle` can do one of four different things, depending on which mode you supply as a command line argument. I have implemented two of these modes for you (`-d` and `-g`), which will help you as you implement the remaining two modes (`-c` and `-x`). Let's look at these modes and what they do.

2.2 Compress Mode (-c)

When `rle` is invoked with with the `-c` argument and provided a filename, for example:

```
$ ./rle -c foo.txt
```

the `rle` program should produce a new output file named after the input file but with the added **file extension** `.rle` – in this example, the file `foo.txt.rle` would be created (or overwritten if it already existed). The contents of the output file is the run-length encoded bytes of the specified input file, in this example that would be the run-length encoded representation of the data in the input file `foo.txt`. The first 4 bytes of the output file must be the Magic Number `21 52 4c 45` (hexadecimal).

When invoked in this fashion with the `-c` mode flag and a filename, execution will fall to the function:

```
void compress(const char *filename);
```

Your first goal is to implement the definition of this function to provide the specified behavior. Your implementation should be able to run-length encode (i.e. "compress") any given file (either binary or text) without limitation on the length of repeating byte sequences in the input file.

2.3 Expand Mode (-x)

When `rle` is invoked with with the `-x` argument and provided a filename, for example:

```
$ ./rle -x foo.txt.rle
```

it should first check that the file ends with the `.rle` file extension and contains the Magic Number `21 52 4c 45` in the first 4 bytes of the file. If either of these requirements are not met, the `rle` program must print the following to `stderr` and terminate without creating or modifying any files:

```
error -- file is not an RLE file!
```

If the specified input file does have the `.rle` file extension and the Magic Number `21 52 4c 45` as the first 4 bytes, then the `rle` program should create a new output file named after the input file but without the `.rle` file extension – in this example, the file `foo.txt` would be created (or overwritten if it already existed). The contents of the output file is the decoded run-length representation of the data in the specified input file, in this example that would be the decoded representation of the data in the input file `foo.txt.rle`.

It is important to note that expanding a previously compressed file should produce the original file. For example:

```
$ echo 'Hello World!!!!!' > foo.txt     (creates foo.txt, contains "Hello World!!!!!")
$ ./rle -c foo.txt                      (produces foo.txt.rle)
$ rm foo.txt                            (deletes foo.txt)
$ ./rle -x foo.txt.rle                  (produces foo.txt, contains "Hello World!!!!!")
$ cat foo.txt                           (print contents of foo.txt to the screen)
Hello World!!!!!
```

When invoked in this fashion with the `-x` mode flag and a filename, execution will fall to the function:

```
void expand(const char *filename);
```

Your second goal is to implement the definition of this function to provide the specified behavior. Your implementation should be able to decode (i.e. "expand") the contents of any properly run-length encoded input file.

2.4 Debug (-d)

I have provided this mode to help you debug your implementations of `compress()` and `expand()`.

When `rle` is invoked with with the `-d` argument and provided a filename, for example:

```
$ ./rle -d foo.txt
```

it will print the raw contents of the specified input file to the screen in two representations – each byte will be displayed in both hexadecimal and ASCII representations. Using our previous example:

```
$ echo 'Hello World!!!!!' > foo.txt
$ ./rle -d foo.txt
00000000: 4865 6c6c 6f20 576f 726c 6421 2121 2121  Hello World!!!!!
00000010: 0a
$ ./rle -c foo.txt
$ ./rle -d foo.txt.rle
00000000: 2152 4c45 0148 0165 026c 016f 0120 0157  !RLE.H.e.l.o. .W
00000010: 016f 0172 016c 0164 0521 010a            .o.r.l.d.!..
$ rm foo.txt
$ ./rle -x foo.txt.rle
$ ./rle -d foo.txt
00000000: 4865 6c6c 6f20 576f 726c 6421 2121 2121  Hello World!!!!!
00000010: 0a
```

Take a few minutes to study the raw bytes displayed in this simple example. You should be able to clearly see how the Magic Number has been added to the compressed file `foo.txt.rle`. Don't stop studying this example until you can clearly see how the the byte sequence `02 6c` is the compressed version of the "double l" in the word "Hello" and that `05 21` encodes the string of exclamation points.

2.5 Generate (-g)

I have provided this mode to help you debug your implementations of `compress()` and `expand()`.

When `rle` is invoked with with the `-g` argument and provided a filename, for example:

```
$ ./rle -g foo.bin
```

the program will write a binary test file that is useful for debugging to disk with the filename specified – in this example, the file `foo.bin` will be created (or overwritten if it already exists).

The contents of this test file can be easily examined using the debug mode flag (`-d`):

```
$ ./rle -g foo.bin
$ ./rle -d foo.bin
00000000: ee5d 4141 418e 2a3a 3a3a 3ae8 bd22 880a  .]AAA.*::::.."..
00000010: acac ac9a 513a 3a3a aaaa aa31 6127 a77e  ....Q:::...1a'.~
00000020: 6d52 d874 6a6f 6e4e e890 a07d 8864 1221  mR.tjonN...}.d.!
00000030: dbae f7f7 f7f7 f7f7 f7f7 f7f7 f7f7 f7f7  ................
00000040: f7f7 f7f7 f7f7 f7f7 f7f7 f7f7 f7f7 f7f7  ................
00000050: f7f7 f7f7 f7f7 f7f7 f7f7 f7f7 f7f7 f7f7  ................
00000060: aaaa bbbb                                ....
```

as you can see, this binary test file has a fair number of repeated and non-repeating byte sequences that will be useful when writing and debugging your `compress()` and `expand()` functions.

Once you have your compress mode (`-c`) is working correctly, you should get the following output when displaying the content of the compressed test file using the debug mode (`-d`):

```
$ ./rle -g foo.bin
$ ./rle -c foo.bin
$ ./rle -d foo.bin.rle
00000000: 2152 4c45 01ee 015d 0341 018e 012a 043a  !RLE...].A...*.:
00000010: 01e8 01bd 0122 0188 010a 03ac 019a 0151  ....."........Q
00000020: 033a 03aa 0131 0161 0127 01a7 017e 016d  .:...1.a.'...~.m
00000030: 0152 01d8 0174 016a 016f 016e 014e 01e8  .R...t.j.o.n.N..
00000040: 0190 01a0 017d 0188 0164 0112 0121 01db  .....}...d...!..
00000050: 01ae 2ef7 02aa 02bb                      ........
```

...and expanding the compressed test file `foo.bin.rle` should, of course, produce the contents of the original file prior to compression:

```
$ rm foo.bin
$ ./rle -x foo.bin.rle
$ ./rle -d foo.bin
00000000: ee5d 4141 418e 2a3a 3a3a 3ae8 bd22 880a  .]AAA.*::::..".. 
00000010: acac ac9a 513a 3a3a aaaa aa31 6127 a77e  ....Q:::...1a'.~
00000020: 6d52 d874 6a6f 6e4e e890 a07d 8864 1221  mR.tjonN...}.d.!
00000030: dbae f7f7 f7f7 f7f7 f7f7 f7f7 f7f7 f7f7  ................
00000040: f7f7 f7f7 f7f7 f7f7 f7f7 f7f7 f7f7 f7f7  ................
00000050: f7f7 f7f7 f7f7 f7f7 f7f7 f7f7 f7f7 f7f7  ................
00000060: aaaa bbbb                                ....
```

Hopefully, you can see how it is sometimes useful to write your own debugging tools. I have written them for you this time, but in the future I won't be there and you will need to write them for yourself. The ability to recognize what debugging tools should be created to help solve a particular problem (or ease development) is an incredibly valuable skill – perhaps even more valuable than the ability to actually develop the tools themselves.

3 Deliverables

Your deliverable for this project will be a ZIP File containing your source code for the `rle` program that meets the above mentioned specifications. You should only submit code that *you* wrote – no copying from your friends, lab mates, the Internet, random people, orcs, Nazgûl, death eaters, Voldemort, Fire Nation troops, dementors, Akatsuki, or cats.

Please read the submission instructions for Project 2 on BBLearn, which describes how to download your project source code as a ZIP File from repl.it and upload it to BBLearn.