Software Engineering 265 Software Development Methods Fall 2019

Assignment 2 Due: Wednesday, October 30, 9:00 pm by submission via git (no late submissions accepted)

Objectives of this assignment

- Use the C programming language to write your implementation of the Lempel-Ziv-Welch compression/decompression algorithm, in a file named “LZW.c”
- Test your code against the provided test data, and other test data you create, to ensure it’s working as expected. The instructor will also be testing your program against test data that isn’t provided to you. Think creatively about how to “break” and strengthen your code!

This assignment: “LZW.c”

In this assignment, we will implement Lempel–Ziv–Welch (LZW), a universal, lossless data compression algorithm. LZW is the algorithm of the Unix file compression utility, compress, and is used in the GIF image format. “Lossless” means that the compression operation preserves 100% of the information in the original format: with a correctly implemented algorithm, you can always recover the identical original data given its compressed form, even though the compressed form *may* be significantly smaller.

While the (glory) days when you could develop a company based on a compression algorithm are long-gone, there is still lasting practical value in knowing how to manipulate files of all types while implementing algorithms of increasing complexity, not to mention continuing to gain proficiency with array-based implementations -- we can achieve all of the above with a deeper dive into compression via LZW.

Broadly speaking, compression is a process of recognizing recurring patterns in some piece of information to be communicated, and replacing multiple duplicate patterns with shorter references to entries in a list of such patterns. For example, imagine you have a list of numbers on a piece of paper, and you need to read it to a friend over the phone. As you’re going through the list, if you see the same number repeated enough times, you’re likely to say something like “48, 3011 ten times, 189, …” and rely on a shared understanding of natural language grammar to distinguish between the in-band data (the numbers) and the out-of-band data (the instructions about patterns). Unfortunately, digital data doesn’t have a “universal grammar” -- general-purpose data compression algorithms that are expected to operate on any type of data must be able to learn the repeated sequences in particular files automatically. As in life, “There ain't no such thing as a free lunch”. For files that contain random data, or patterns that are too diverse or too sparsely distributed, a general-purpose compression algorithm might even produce “compressed” data that’s larger than the original!

LZW is a method for adaptive / dynamic compression - it starts with an initial model, reads data piece by piece and then updates the model while encoding the data as it proceeds through the input file. LZW can be used for binary and text files - it works on any type of file -- that said, it is

known for performing well on files or data streams with (any type of) repeated substrings (e.g. text files, which often contain frequently used words like “an”, “the”, “there”, etc.).

With English text, the algorithm can be expected to offer 50% compression levels or greater. Binary data files, depending on structuring are capable of decent compression levels as well - for this assignment, file size is not part of the grading criteria--in fact, the evaluation relies on your code producing exact replicas of the expected compressed files so don’t get *too* creative!

As in the previous assignment, we are interested in “round-trip” encoding and decoding, translating from the original format to encoded and back to original. I hope that some of the lessons from the previous assignment will become useful sources for your reuse. While the compression technique may be new, other details remain similar. Implementing a more complex algorithm shows how we might layer on complexity once we establish base skills.

Background Information

Dictionary Data Structure

A **`dictionary`** is a general-purpose data structure for storing a group of objects. A dictionary has a set of keys and each key has a single associated value - the elements in this abstract data structure are commonly called, **`key-value pairs`**. Dictionaries are usually implemented using hash tables, and are often known as “dict”, “hash”, “map”, or sometimes “hashmap” in different programming languages.

Binary Coding Schemes

The “**`LZW`**” algorithm described by Welch's 1984 article [1] encodes variable-length sequences of 8-bit bytes as fixed-length 12-bit codes.

As an example of a binary code, this toy coding scheme uses three-bit codes to represent a choice of eight possible symbols (the letters ‘A’ through ‘H’). The choice of symbols is arbitrary--we could have used eight different emoji. As long as the sender and receiver have a shared understanding of what the codes mean, they can communicate using compressed data.

binary code	**symbol**
000	A
001	B
010	C
011	D
100	E
101	F

110	G
111	H

In a binary code, each additional bit added to the code scheme doubles the number of distinct symbols that can be encoded--for an n-bit code, 2^n different symbols can be represented. In the classic LZW algorithm, the choice of a 12-bit coding scheme allows for 4096 distinct symbols with binary codes between 000000000000 and 111111111111. Each binary code can also be represented as three hexadecimal digits, e.g. 000 to FFF.

The dictionary data structure to be used for this assignment has a capacity of 4096 entries (2^{12}), of which:

- the first 256 entries, numbered 0 through 255, are reserved for every possible 1-byte value (i.e. entry #0 is for the ASCII NUL character, entry #32 is for the ASCII space character, entry #65 is for the capital letter “A”... look at **`man ascii`**! (press ‘q’ to exit the man page reader))
- the entries numbered 256 through 4094 are assigned dynamically by the compression algorithm to represent multi-byte sequences in the input data.
- the last entry, numbered 4095, is reserved for a padding marker

The code that refers to an entry in the dictionary is simply the entry’s numeric position, encoded as a 12-bit binary integer.

Additional Notes

Both encoding and decoding programs refer to the dictionary countless times during the algorithm. You may want to reference the hints in the Provided Materials section of this document.

The encoder looks up indexes in the dictionary by using strings. That is, to aid the look-up of indexes, we use a dictionary of strings - i.e. an array of strings. With each string having some maximum number of characters, you might consider using as your data structure, a multidimensional array -- as discussed in Unit 3, Slide 29, this has a form similar to: e.g., X[row][column], recall that we access elements of the array through integer indexes.

Why select such an approach? A data structure with 0(1), “Order-1 time complexity”, can be useful in practice.

Algorithm and Example

LZW Encoding

---

1. Start with a *standard initial dictionary* of 4096 entries (each entry can be identified using a 12-bit number)
2. Set *w* to `' '`
3. As long as there are more characters to read
    a. read a character *k*
    b. if *wk* is in the dictionary
        set *w* to *wk*
      else
        output the code for *w*
        add *wk* to the dictionary, if it's not full
        set *w* to *k*
4. output the code for *w*

Note #1: Step 2 is:
  Set w to `' '`
It is not:
  Set w to `'  '`
That is, you start off with w set to nothing, not a space character. So if you append 't' to '', you end up with 't' (not ' t').

Note #2: The first 256 entries (0-255) in the LZW dictionary should be the 1-byte codes whose values are 0-255. That is,
  dict[42][1] is 42 (and dict[42][0] is 1)
  dict[228][1] is 228 (and dict[228][0] is 1)
In general, for 0 <= i <= 255: dict[i][1] is i (and dict[i][0] is 1)

LZW Decoding

---

1. Start with a *standard initial dictionary* of 4096 entries (the first 256 are standard ASCII, the rest are empty).
2. Read a code *k* from the encoded file
3. Output dict[*k*]
4. Set *w* to dict[*k*]
5. As long as there are more codes to read in the input file
    a. read a code *k*
    b. if *k* is in the dictionary
        output dict[*k*]
        add *w* + first character of dict[*k*] to the dict
      else
        add *w* + first char of *w* to the dict *and* output it
    c. Set *w* to dict[*k*]

The programs LZW.c takes as its first two inputs, the name of an input file as the first command line argument, and the name of an output file as the second command line argument.

If the second passed command line argument is an ‘e’, then the program creates a new file with the same name as the file in the command line, but with the extension .LZW. The new file contains the LZW encoding of the original file.

If the second passed command line argument is a ‘d’, then the program takes the name of compressed file as a command line argument (a file ending in extension, .LZW). The program interprets the original file name by dropping the .LZW file extension and then decodes the LZW encoding back to the “original.” (actually a copy).

The program should open both the input file and output file as binary files and make no assumption about what byte values appear in the files. Here's what their behaviour looks like:

```
~ /assignment2 > cat tests/encode_short_string/input.txt
this_is_his_history

~ /assignment2 > ./tools/b2x ./tests/encode_short_string/input.txt
746869735F69735F6869735F686973746F72790A

~ /assignment2 > ./LZW ./tests/encode_short_string/input.txt e

~ /assignment2 > ./tools/b2x ./tests/encode_short_string/input.txt.LZW
07406806907305F10205F10110310707406F07207900AFFF
# different hex code means the input file and the encoded file are different

~ /assignment2 > diff ./tests/encode_short_string/compare.txt.LZW
./tests/encode_short_string/input.txt.LZW
# no output means the files are the same

# rename our file so we don’t overwrite it (harder to determine if program executed correctly)
mv ./tests/encode_short_string/input.txt ./tests/encode_short_string/input.old

# now let’s decode our compressed file, and make sure it matches the original input
~ /assignment2 > ./LZW ./tests/encode_short_string/input.txt.LZW d

~ /assignment2 > cat ./tests/encode_short_string/input.txt
this_is_his_history

~/assignment2 > diff ./tests/encode_short_string/input.txt ./tests/encode_short_string/input.old
# no output means they’re the same!

~ /assignment2 >
```

This session shows the contents of the file input.txt. The file contains the string this_is_his_history.

The hexadecimal values of the ASCII codes are also shown. Running the program LZW (with the encode flag) on input.txt produces a file input.ext.LZW. The file input.txt.LZW contains 12-bit values for the LZW codes:

074 = t

068 = h

069 = i

073 = s

05F = _

102 = is

05F = _

101 = hi

103 = s_

etc.).

The decode version of the program, LZW <filename> d, then correctly recovers the original file name from the given input, dropping .LZW from the input file name, and recovers the original data.

Requirements

The requirements for this assignment are as follows:

1. Use arrays (and not dynamic memory allocation!) as an aggregate data type for file processing (encoding and decoding).
2. Create a **main** function that:
    a. Takes two command line arguments:
        i. A string indicating the name of the file to operate on (the “input” file);
        ii. A one-character flag indicating whether the program should operate in encoding or decoding mode. Use ‘e’ for encoding mode, or ‘d’ for decoding mode.

    If the second argument is not provided, or is not either ‘e’ or ‘d’,
        iii. print “Invalid Usage, expected: LZW {input_file} [e | d]” to the console
        iv. terminate the program with exit code 4.

    b. Opens the input file indicated by the first argument.

    If there is no filename specified:
        i. print “Error: No input file specified!” to the console

ii. terminate the program with exit code 1.

If the input filename specified doesn't exist or can't be read for any reason:

iii. print "Read error: file not found or cannot be read" to the console

iv. terminate the program with exit code 2.

c. Opens an output file using the naming convention as specified

d. Depending on the value of the second command-line argument, passes the string read from the input file to either the **encode** or **decode** function, as described below

3. Create a second function called **encode** with the following characteristics:
   a. Takes two C files as function parameters:
      i. an input file
      ii. an output file
   b. Encodes the provided file using the algorithm described above.
   c. If the file was encoded successfully:
      i. create the encoded representation to the file system
      ii. terminate the program with exit code 0, indicating success
   d. If the file could not be encoded for any reason (e.g. a particular input fails):
      i. print "Error: File could not be encoded" to the console
      ii. terminate the program with exit code
4. Create a third function called **decode** with the following characteristics:
   a. Take two C files as function parameters:
      i. an input file
      ii. an output file
   b. Decode the provided file using the algorithm described above

      If your program encounters invalid, corrupted or badly formatted data that cannot be decoded according to the specified algorithm, the program should:

      i. print "Error: Invalid format" to the console

      ii. terminate the program with exit code 3.

   c. If the file was decoded successfully:
      i. write the resulting decoded representation to a file with the expected name
      ii. terminate the program with exit code 0, indicating success
   d. If the file could not be decoded for any reason (other than those stated above):
      i. print "Error: File could not be decoded" to the console

ii. terminate the program with exit code 5

By now, you're developing your own tests to evaluate your software (we don't get what we *expect*, we get what we ***inspect***) -- consequently, we would like to see at least one test input from YOU - use the testing convention described in this assignment - i.e. if your test input is expected to successfully execute, then it should follow the happy path of the algorithm. If the test input is expected to fail, then it should show a brief error message followed by exiting with error code ***6***.

You may also think of tweaks to the algorithms that will make the encoder/decoder run faster, or make the compression better. If so, implement these ideas, and compare. But, this is not necessary. You are encouraged to devote time to testing your program.

Provided Materials

We have created a git repository for you - it contains a starter file, LZW-starter.c, a tool, b2x, a limited test suite, and these assignment instructions. There are also some hints in this section.

LZW-starter.c

A "starter kit" for LZW.c for you to reuse in your implementation. Copy it to LZW.c and edit it, or copy and paste parts of it. The starter-file contains a few functions you are invited to use in your solution - these are read12(), write12() and flush12(). It is not your aim to understand the implementation of these functions (although you're welcome to explore as you see fit) - it IS your aim to understand how you might apply. these functions in your solution - if you choose to apply them, it should have the effect of simplifying the solution. The use of flush12() might have an analogous application to the use of fflush(stdin) - while we're on the subject: fflush() might have some use in your solution as well.

b2x Tool

We created a directory /tools and placed a source file, b2x.c, that reads a binary file and writes the "binhex" representation of a file to the screen -- binhex is short for binary-to-hexadecimal - it's yet another encoding mechanism - you can compile and use this tool if useful (it is not required). You can also use the common unix hexdump utility for similar purposes.

Test Suite

We will supply a limited suite of test cases that you can use to evaluate your own program, and get a feel for the types of inputs you can expect. You will see the tests include both text and binary files. The test suite has the same structure as the previous assignment - a set of directories, with one test case provided per directory. The directory, as previously, is named according to the type of test - e.g. encode_text or decode_image, where the first word denotes the mode the program should be run in while running this test.

Things are not identical to the previous assignment - for example, in the case of encoding:

- the filename passed may have *any* name and extension, e.g. **`input.ext`** (your program is expected to read that file as its input, with ext replaced, e.g. with “txt” or “jpg”)
  NOTE: Your program **does not** need to locate the appropriate file to operate on -- you can look at the files in the test case directory, and pass the input filename as a command line parameter!
- Continuing the example above, if your file is expected to run successfully given its input, the directory will contain a file whose name is ***`compare.ext`*`.LZW`** where ***`ext`*** is the original uncompressed filename extension
  - NOTE: Why “compare.ext”? -- Well, we give you the input file for testing purposes; if properly run, you should get input.ext.LZW as your output - if we provided you with this properly named file as output, then you could perhaps overwrite it accidentally, which would remove your ability to compare - hence, we prefix the outputs with “compare”.
- Your program’s output is expected to *precisely* (bit-for-bit!) match the contents of this output file.
- If your program is expected to terminate with a non-zero (unsuccessful) error code when given the provided input file, the directory will contain one of the following:
  - a file named ***`n`*`.err.txt`**, where ***n*** is the expected exit code, containing text that your program’s output is expected to precisely match, **or;**
  - a file named ***`n`*`.err`**, where ***n*** is the expected exit code. In this case your output doesn’t need to match any specific error message, but your program should still exit with error code ***n***
  - NOTE: for YOUR test(s) that you provide, should the test(s) exit with an error code, the error code must be ***6***.
- In the case of decoding, we provide input.ext.LZW and also provide compare.ext - again, you can derive the resulting output filename from the given input, and then compare accordingly.

Hints

- Use an array of strings for the dictionary. Here's an example:
  unsigned char dict[4096][32];
- That's an array of 4096 strings, with each string having a maximum of 32 characters. That means that your program must be careful not to add to the dictionary once it is full and not to add strings to the dictionary that are too long. The code for each entry in the dictionary is simply its position in the array. For example, the code for the entry dict[267] is 267

- I strongly recommend against using the standard C null-terminated strings in the dictionary (since null might be a valid character in a LZW dictionary entry). Instead, store the length of the string as the first byte of the string like this:
  dict[i][0] = 3;
  dict[i][1] = 'a';
  dict[i][2] = 'b';

```
dict[i][3] = 'c';
```

You will not be able to use the built-in string functions (like strcmp, strlen, etc.) for this type of string.

Process

To get started, the assignment repository has already been set up in your person GitLab space, listed as /assignment2.git. We set this up in our local machine space with **`git clone`**:

```
git clone
https://gitlab.csc.uvic.ca/courses/2019091/SENG265/assignments/<NETLIN
K_ID>/assignment2.git
```

(all one line, with <NETLINK_ID> replaced with your personal Netlink identifier)

Note that by cloning from your personal repository, this automatically sets up your remote on GitLab so when you **`git push`** later, your changes will automatically be visible to both you and the teaching team.

Use your favourite text editor to write your program.

Once you're able to compile your program, test it by running it with the **`input.<ext>`** files in the provided test suite, and any other files you can think of. Be creative; try to predict what kinds of inputs will break your code! Try random files lying around on your computer or the internet to see what happens!

If it seems like your changes are having no effect, don't forget to recompile your program after changing it.

When you've made changes to the file that you want to keep, **`git commit`** your changes to your local copy of the repository. Git will save the history of past commits. When you want to upload your changes to your repository, use **`git push`**.

Deliverables

- Write a C program in a file named LZW.c (case sensitive), placed in the root of your assignment2 repository, that implements the above requirements.
- The program must compile successfully using **gcc** (within the lab environment)
- We will compile with **`gcc LZW.c -o LZW`**
- The assignment will be marked based on the last version of your program pushed before the deadline.

Evaluation

Compilation

(10% of the mark, if your program compiles at all)

Correctness

(60% of the mark, based on the proportion of provided and secret test cases that your program successfully passes)
The teaching team will run your code using the provided test suite to verify whether your program meets the requirements:

- When run with correct parameters and given input in the expected format, it produces the expected output
- When run with incorrect parameters or badly formatted input data, it produces the expected error messages and the expected exit codes

We will also evaluate your program using ***an additional test suite that is not provided to you***, to ensure that your solution isn't over-fit to the provided test suite.

Your Test Suite

(10% of the mark, YOU supply additional test(s) constructed using the same scheme)

The teaching team will run your test case against your solution and validate that it works. We might also run your test case against *other* student solutions. If seen as a particularly effective test, the teaching team reserves the right to provide a small bonus grade beyond the 10% allocated for this item.

Style

(20% of the mark, based on the subjective assessment of the teaching team)

The teaching team will read your code to judge whether it is clean, original, elegant and fit to purpose.

A few examples of bad programming style:

- **inconsistent formatting**, indentation, etc. -- pick a style and use it consistently
- **misleading or non-descriptive names** for variables and functions
- **insufficient whitespace** impairing readability
- long sections of **repetitive code** (sometimes it's better to repeat short sections than introduce an abstraction!)
- **"dead" code**, i.e. present in the file but never called
- **complex code without explanatory comments** (don't go overboard; simple code often doesn't need comments)

Programming environment

For this assignment, please ensure your work compiles and executes correctly on the Linux machines in ELW B238. You are welcome to use your own laptops and desktops for much of your programming; if you do this, **give yourself a few days before the due date to iron out any bugs in the C program you have uploaded to the BSEng machines**. (Bugs in this kind

of programming tend to be platform specific, and something that works perfectly at home may end up crashing on a different hardware configuration.)

Individual work

This assignment is to be completed by each individual student (i.e., no group work). Naturally you will want to discuss aspects of the problem with fellow students, and such discussion is encouraged. However, sharing of code fragments is strictly forbidden without the express written permission of the course instructor (Murray). If you are still unsure regarding what is permitted or have other questions about what constitutes appropriate collaboration, please contact me as soon as possible. (Code-similarity analysis tools will be used to examine submitted programs.)

Additional Information

Because image compression remains a subject of great interest, you may wish to learn more about it -- feel free to review or ignore the following content as you see fit:

YouTube videos on image file compression

- JPEG 'files' & Colour (JPEG Pt1)- Computerphile
- JPEG DCT, Discrete Cosine Transform (JPEG Pt2) Computerphile

Articles on image file compression

- TIFF LZW Compression Algorithm: https://www.fileformat.info/format/tiff/corion-lzw.htm
- GIF LZW Compression Algo: http://giflib.sourceforge.net/whatsinagif/lzw_image_data.html

References

[1] A Technique for High Performance Data Compression, IEEE Computer, Terry A. Welch, June 1984 (8:19), accessed at this link
[2] LZW Data Compression, Dr. Dobbs Journal, Mark Nelson, October 1989, accessed at this link
[3] LZW Data Compression Revisited, Dr. Dobbs Journal, Mark Nelson, November 2011 at link