Assignment 5: Data Compression

by Dr. Kerry Veenstra
CSE 13S, Fall 2024
Document version 1 (changes in Section 6)

Due Wednesday November 20th, 2024, at 11:59 pm
Draft Due Monday, November 18th, 2024, at 11:59 pm

1 Introduction

In this assignment you are going to write two programs and an Abstract Data Type (ADT). The two programs are a data compressor and a corresponding decompressor. The ADT will make it easier to read and write the compressed data files. You will be given unit tests for the ADT.

The compression program, `compress`, will use a simple algorithm that is similar to the one that was used in the original Apple Macintosh *MacWrite* program to decrease the sizes of word-processing files[1]. Here is an example of using `compress` to reduce the size of the nonsense poem "Jabberwocky":

```
$ ./compress jabberwocky.txt -o jabberwocky.comp  Enter      ⟵ compress jabberwocky.txt
$ ls -l jabber*  Enter      ⟵ compare the sizes of the original and compressed files
-rw-rw-r- 1 ubuntu ubuntu 814 Nov 11 13:40 jabberwocky.comp      ⟵ size of compressed file
-rw-rw-r- 1 ubuntu ubuntu 1017 Nov 11 13:31 jabberwocky.txt      ⟵ size of original file
```

The compressed file has `814` bytes, while the original file has `1017` bytes. So the compressed file is $814 \div 1047 = 78\%$ of the original size.

The decompression program, `decompress`, reverses the compression process. You can compare the output of the decompressor with the original, uncompressed file.

```
$ ./decompress jabberwocky.comp -o jabberwocky.decomp  Enter      ⟵ decompress
$ diff -s jabberwocky.txt jabberwocky.decomp  Enter      ⟵ compare result with original
Files jabberwocky.txt and jabberwocky.decomp are identical      ⟵ Success!
```

You also will be given unit tests for the ADT and a shell script to run system tests.

2 File I/O

All of this assignment's programs will read and write text files, and all of them will use the same set of File I/O routines.

2.1 Example: Reading a File, Byte-by-Byte

A program can read a file one character (or byte) at a time with the `fgetc()` function. Before calling `fgetc()`, a program must open the file with `fopen()`, and then when finished reading the input file, a program must close it with `fclose()`. The `fgetc()` function lets you know that it has returned all of the characters of the input file by returning `EOF` ("end of file").

Be sure to store the fgetc() return value in a variable of type int. Although fgetc() can return $256 = 2^8$ different char values, which fit in a char, it also returns a 257^{th} value: EOF. Since a char cannot store 257 different values, use an int variable to hold the fgetc() return value.

print_file_example.c is an example of using these functions to read data from a file and to print it to the screen.

```
void print_file_example(const char *filename) {   ⟵ The function is called with a file name.
    FILE *f = fopen(filename, "r");          ⟵ Open file for reading ("r"). Save return value f.

    if (f == NULL) {              ⟵ Check fopen() return value. It's NULL? Report an error.
        printf("Can't open file for input:  %s\n", filename);
        exit(1);
    }

    while (1) {                   ⟵ Read the characters of the file in a loop.
        int ch = fgetc(f);        ⟵ Get the next char from the file. fgetc() returns an int.
        if (ch == EOF) break;     ⟵ Leave the loop if fgetc() returns EOF ("end of file").

        // Print ch, but here is where the program could write to an output file.
        printf("%c", ch);         ⟵ Print the char.
    }

    fclose(f);      ⟵ We are done reading the file. Close it.
}
```

2.2 Example: Writing a File, Byte-by-Byte

In addition to *reading* file data one character (or byte) at a time using fgetc(), a program can *write* file data one character at a time using fputc(). Before calling fputc(), a program must open the file with fopen(), and after writing the last char of the output file, a program must close the file with fclose().

If the file already exists, then calling fopen(filename, "w") will erase the contents of the file to prepare it for new characters from fputc(). (It is possible to "append" characters to a file without first erasing its contents, but we don't do that in this assignment. See man 3 fopen and "a" mode for more information.)

write_file_example.c is an example of using these functions to write a character string to a file.

```
void write_file_example(const char *filename) {   ⟵ The function is called with a file name.
    FILE *f = fopen(filename, "w");          ⟵ Open file for writing ("w"). Save return value f.

    if (f == NULL) {          ⟵ Check fopen() return value. It's NULL? Report an error.
        printf("Can't open file for output:  %s\n", filename);
        exit(1);
    }

    char string[] = "Write this string to the file.\n";
    char *p = string;         ⟵ Prepare to walk through the string. Now *p == 'W'.

    while (*p != '\0') {      ⟵ Walk through the chars of the string.
        fputc(*p, f);         ⟵ Write a char of the string to the file using f.
        ++p;                  ⟵ Point to the next char in the message.
    }

    fclose(f);      ⟵ We are done writing the file. Close it.
}
```

3 Nibble-by-Nibble

The data compression algorithm that will be presented in Section 4 writes 4-bit chunks of data instead of writing full 8-bit bytes. The computer-science term for a 4-bit chunk of data is "nibble." The $16 = 2^4$ possible values of a nibble range from binary 0000_2 through 1111_2 (decimal 0 through 15). The eight bits of a byte can be split into two nibbles, often called the "high nibble" and the "low nibble."

7	6	5	4	3	2	1	0	⟵ bit number of byte
High Nibble				Low Nibble				
Byte or `char`								

The file `nibbler.h` defines a struct and four functions that you can use to read and create files one nibble at a time. The struct, called `NIB`, manages the reading and writing of nibbles. You use `NIB` for nibble-based file I/O very much as you use `FILE` for character-based file I/O (Section 2). The table below compares the functions of `stdio.h` and the functions of `nibbler.h`. You will be writing the functions that are defined in `nibbler.h`.

	Byte I/O	⟶	Nibble I/O
Include File	`#include <stdio.h>`	⟶	`#include "nibbler.h"`
Data Declaration	`FILE *f;`	⟶	`NIB *nib;`
Opening a File	`f = fopen(filename, mode);`	⟶	`nib = nib_open(filename, mode);`
Reading Data	`char ch = fgetc(f);`	⟶	`int nibble = nib_get_nibble(nib);`
Writing Data	`fputc(ch, f);`	⟶	`nib_put_nibble(nibble, nib);`
Closing a File	`fclose(f);`	⟶	`nib_close(nib);`

Internally, the Nibble-I/O functions in the table above will call traditional Byte-I/O functions as needed. They will use the `NIB` data type defined below to guide their operation. For example:

- `nib_open()` will allocate a `NIB` data structure, call `fopen()`, and then initialize fields of the `NIB` data structure. See comments in `nibbler.c` for more details.
- `nib_get_nibble)` will read a `char` using `fgetc(`, **store** the **least-significant nibble** in the `NIB` struct (as an integer value 0 through 15), and **return** the **most-significant nibble** as an integer value 0 through 15. However, if `nib_get_nibble()` already has a stored nibble (from an earlier call), then instead of calling `fgetc()` it will return the stored nibble. See comments in `nibbler.c` for more details.
- `nib_put_nibble()` will store a nibble in `nib->stored_nibble`, set `nib->num_nibbles = 1`, and wait to be called with a second nibble. Then it will merge the two nibbles into a `char` that it writes using a call to `fputc()`. After which it will set `nib->num_nibbles = 0`. See comments in `nibbler.c` for details.
- `nib_close()` will close the file by calling `fclose()`, however if the `NIB` data structure is storing a nibble, then before closing the file, `nib_close()` will write that nibble to the file as the most-significant nibble of a `char` (whose least-significant nibble is 0000_2). See the comments in `nibbler.c` for details.

The C programming language has operators that you can use to isolate the high nibble and the low nibble of a byte. Given an `int x`, then `high_nibble = (x >> 4) & 0xf`, and `low_nibble = x & 0xf`. Also, given a high nibble and a low nibble, then combine them into a byte `x = high_nibble << 4 | low_nibble`. (These equations assume that $0 \leq$ `high_nibble` ≤ 15 and $0 \leq$ `low_nibble` ≤ 15.)

```
typedef struct s_nib NIB;

struct s_nib {
    FILE *underlying_f;     // The result of calling fopen().
    bool opened_for_read;   // true when fopen()'s mode was "r".  false otherwise.
    int num_nibbles;        // 0 or 1
    int stored_nibble;      // Use the value when num_nibbles == 1.
};
```

4 Data Compression

The characters of normal text files all have the same size (1 byte). To compress such files, your compression program relies on some characters being more common than others. Each common character is converted into 1 nibble, while less common characters are converted into 3 nibbles. On average, when compressing modern English text, the compressed file will be smaller than the uncompressed file.

The 15 specific characters in the list below are considered to be "common." Each of these characters will be represented in a compressed-data file by a different nibble (see Fig. 2). Everything else will be represented by a unique sequence of 3 nibbles:

$$\begin{aligned} \text{a, c, d, e, f, h, i, l, n, o, p, r, s, t, } \textit{space} &\rightarrow 1 \text{ nibble} \\ \text{all others} &\rightarrow 3 \text{ nibbles} \end{aligned}$$

Consider the 16-character phrase "computer science". Each of the 14 red underlined characters is converted using the table in Fig. 2. For example, `'c'` $\rightarrow 13 = \texttt{d}_{16}$. The two characters that are not in the table are represented by their hex value with a `0` prefix: `'m'` $\rightarrow \texttt{6d}_{16} \rightarrow \texttt{06d}_{16}$ and `'u'` $\rightarrow \texttt{75}_{16} \rightarrow \texttt{075}_{16}$. The resulting nibble sequence (in hexadecimal) is `d606df07532519d824d2`. The total size of the compressed data is:

$$14 \times 1 \text{ nibble} + 2 \times 3 \text{ nibbles} = 20 \text{ nibbles} = 10 \text{ bytes}$$

The size of the compressed data is $10 \div 16 \approx 63\%$ of the original data, for an overall reduction in space.

4.1 Data Compression Algorithm

As described above, the *MacWrite* word processor reduced 15 of the possible byte values to 1 nibble, and it expanded the remaining 241 byte values to 3 nibbles. Our version of the algorithm is in Fig. 1.

Use `fgetc()` to read data from the input file one byte at a time, and use the `nibbler` ADT in Section 3 to write compressed data one nibble at a time.

```
compress():
    loop:
        Use fgetc() to get an int ch.
        If ch is EOF then break
        If ch is in the table in Fig. 2.
            Convert ch into its corresponding nibble from Fig. 2.
            Write the nibble using nib_put_nibble().
        else
            Write a 0 nibble using nib_put_nibble()
            Write the high nibble of ch using nib_put_nibble()
            Write the low nibble of ch using nib_put_nibble()
```

Figure 1: `compress()` algorithm. Uses table in Fig. 2.

`' '` $\rightarrow 1$	`'o'` $\rightarrow 6$	`'l'` $\rightarrow 11$
`'e'` $\rightarrow 2$	`'a'` $\rightarrow 7$	`'h'` $\rightarrow 12$
`'t'` $\rightarrow 3$	`'i'` $\rightarrow 8$	`'c'` $\rightarrow 13$
`'n'` $\rightarrow 4$	`'s'` $\rightarrow 9$	`'f'` $\rightarrow 14$
`'r'` $\rightarrow 5$	`'d'` $\rightarrow 10$	`'p'` $\rightarrow 15$

Figure 2: Lookup table for converting a byte into a nibble. Used by the `compress()` algorithm of Fig. 1.

4.2 Data Decompression Algorithm

Decompression is the reverse of compression. Our decompression algorithm is in Fig. 3. Use the `nibbler` ADT in Section 3 to read compressed data one nibble at a time, and use `fputc()` to write uncompressed data to the output file one byte at a time.

To decompress the nibble sequence `d606df07532519d824d2`, look up the single, non-zero nibbles in the table of Fig. 4. So to start, $\mathtt{d}_{16} = 13 \rightarrow$ `'c'` and $\mathtt{6}_{16} = 6 \rightarrow$ `'o'`. Next is a zero-value nibble, which identifies a three-nibble sequence. Convert $\mathtt{06d}_{16}$ by throwing away the `0` and combining each remaining pair of nibbles as a character: $\mathtt{06d}_{16} \rightarrow \mathtt{6d}_{16} \rightarrow$ `'m'`. Finish the conversion, and the decompressed result is `computer science`.

4.3 The Inconvenient Reality of Data Compression

Most files of modern English text will compress well using this algorithm, but that's not a guarantee. In fact, for *any* (lossless) data-compression algorithm, we always can create a data file that does not compress—and actually expands. For our algorithm, consider the 5-letter word "buggy". Your compression program considers none of the letters to be "common," and so they all will be represented by 3 nibbles. The resulting data will grow in size:

$$5 \times 3 \text{ nibbles} = 15 \text{ nibbles} = 7\tfrac{1}{2} \text{ bytes}$$

The size of the "compressed" data is $7\frac{1}{2} \div 5 = 150\%$ of the original data, or actually an expansion.

Every data-compression algorithm is designed for the kind of data that it compresses. Consequently, there are as many data-compression algorithms as there are kinds of data. (The book *Data Compression: The Complete Reference, 4th ed.* [1] has over 1000 pages!) For this reason, we will be testing your programs using English-language text files—the kind of data that the *MacWrite* compression algorithm was designed for. But since it will be interesting, we may throw in a few non-English text files to see how the algorithm works.

```
decompress():
   loop
      read a nibble using nib_get_nibble()
      if nibble is EOF then break
      if nibble > 0:
         Look up nibble in the table of Fig. 4
         output the corresponding byte value
      else
         read another nibble as nibble1 using nib_get_nibble()
         if nibble1 is EOF then break
         read another nibble as nibble2 using nib_get_nibble()
         if nibble2 is EOF then break
         create a byte using nibble1 as the high nibble and nibble2 as the low nibble
         output the byte using fputc()
```

Figure 3: `compress()` algorithm. Uses table in Fig. 4.

$1 \rightarrow$ `' '`	$6 \rightarrow$ `'o'`	$11 \rightarrow$ `'l'`
$2 \rightarrow$ `'e'`	$7 \rightarrow$ `'a'`	$12 \rightarrow$ `'h'`
$3 \rightarrow$ `'t'`	$8 \rightarrow$ `'i'`	$13 \rightarrow$ `'c'`
$4 \rightarrow$ `'n'`	$9 \rightarrow$ `'s'`	$14 \rightarrow$ `'f'`
$5 \rightarrow$ `'r'`	$10 \rightarrow$ `'d'`	$15 \rightarrow$ `'p'`

Figure 4: Lookup table for converting a nibble into a byte. Used by the `decompress()` algorithm of Fig. 3.

5 Your Task

5.1 Copy the Resource Files

☐ **To do:** Synchronize the **resources** repository on your laptop's Multipass instance. Then copy the files for asgn5 into your repository.

```
$ cd                                   ⟵ go to your "home" directory
$ cd f24/13s/resources                 ⟵ go to your laptop's copy of the resources repository
$ git pull                             ⟵ synchronize your laptop's copy of the resources repository
$ cd ..                                ⟵ this changes your current directory to ~/f24/13s
$ cp resources/asgn5/* yourcruzid/asgn5      ⟵ copy the files for asgn5
$ cd yourcruzid/asgn5                  ⟵ get ready to run the demo executable and write your program
```

5.2 Submit a draft for design.pdf

☐ **To do:** Read through this document and make notes about what you need to do. In particular, pay attention to sections 4 through 4.3. Also, read through the starter files.
☐ **To do:** Create a draft of your design document, **design.pdf**. The purpose of this draft is to document your initial ideas about the design. Your draft should answer these questions:

1. How does your `compress` program compress a file?
2. Your `compress` and `decompress` programs use conversions that are defined in tables in Fig. 2 (byte value to nibble value) and Fig. 4 (nibble value to byte value). Describe three ways to implement these conversions in a C program.
3. Which method of the prior question do you prefer using, and why?

☐ **To do:** add/commit/push/submit commit ID for the **design.pdf** by Monday, November 18th at 11:30 pm. Note: You can submit this part before the due date. We are asking for a draft. Submit that draft, and the move onto the next steps.

5.3 Make a Makefile

Why do you use a `Makefile`? Because you need to make three executable files: `unittests`, `compress`, and `decompress`.

Unlike some other programming languages, if you have *multiple* C program files, you create an executable file through multiple intermediate "object files" (`nibbler.o`, `unittests.o`, `compress.o`, and `decompress.o`). The steps are shown below. These are what we want the `make` command to do. (Add the `-g` option to the `clang -c` command to make it possible to use a debugger.)

What `make` Does when Compiling Three Executables:

```
clang -c -g nibbler.c                        ⟵ create the "object file" nibbler.o from nibbler.c
clang -c -g unittests.c                      ⟵ create the "object file" unittests.o from unittests.c
clang -c -g compress.c                       ⟵ create the "object file" compress.o from compress.c
clang -c -g decompress.c                     ⟵ create the "object file" decompress.o from decompress.c
clang unittests.o nibbler.o -o unittests     ⟵ create "executable file" unittests
clang compress.o nibbler.o -o compress       ⟵ create "executable file" compress
clang decompress.o nibbler.o -o decompress   ⟵ create "executable file" decompress
```

☐ **To do:** Create the file `Makefile` to make the `unittests`, `compress`, and `decompress` executables.

- **The first two lines of your Makefile are below.** Remember that the `make` program will make the **first** target that is listed in your `Makefile`, along with any other dependent files. (The "target" is underlined in the dependency line below.) So if the first dependency line of a `Makefile` is the one below, then the `Makefile` will execute commands to make `unittests`, `compress`, and `decompress` (which is what you want). If the dependency for `all` is not the first dependency in the file, then typing `make` will not make all of the executables.

  When looking at a `Makefile` dependency line, remember, "this *depends on* that."

  ```
  .PHONY: all        ⟵ tell make that all isn't really a file that gets made
  all: unittests compress decompress
  ```

- Remember that Assignment 3 and the lecture on October 28 demonstrated using a `Makefile`.

☐ **To do:** Run your `Makefile` to compile `unittests`, `compress`, and `decompress`.

Running your `Makefile`

```
$ make      ⟵ That's all that's needed.
```

5.4 Write and Test the Functions of the Nibbler ADT

Look in Section 3 and in `nibbler.c` for the details about the four ADT functions.

☐ **To do:** Write `nib_open()`.
☐ **To do:** Confirm that the first three unit tests pass when you run `./unittests 3`.

If a test fails, then you will need to open `unittests.c` and look at the code on and near the offending line.

☐ **To do:** Write `nib_close()`.
☐ **To do:** Confirm that the first five unit tests pass when you run `./unittests 5`.

☐ **To do:** Write `nib_get_nibble()`.
☐ **To do:** Confirm that the first eight unit tests pass when you run `./unittests 8`.

☐ **To do:** Write `nib_put_nibble()`.
☐ **To do:** Confirm that all 12 unit tests pass when you run `./unittests 12`.

5.5 Write compress.c

☐ **To do:** Write `main()`, which is going to confirm that the command line has these arguments after the program name: "infile `-o` outfile". Then it will open the infile using `fopen()` and open the outfile using `nib_open()` and call `compress()`.
☐ **To do:** Write `compress()`. The algorithm for `compress()` is documented in both Section 4.1 and in the file `compress.c`.

5.6 Write decompress.c

☐ **To do:** Write `main()`, which is going to confirm that the command line has these arguments after the program name: "infile `-o` outfile". Then it will open the infile using `nib_open()` and open the outfile using `fopen()` and call `decompress()`.
☐ **To do:** Write `decompress()`. The algorithm for `decompress()` is documented in both Section 4.2 and in the file `decompress.c`.

5.7 Test Your Program

To receive full credit, the output of your programs must match the output of the reference programs that we provide. You can use the `diff` program as you did in prior assignments to compare the output of the reference program and the output of your program.

As with Assignment 4, we've given you a test script that automates the comparison! The test script is called `runtest.sh`. Here is an example of running the test script.

Testing in a Multipass Shell

```
$ source runtest.sh jabberwocky.txt
```
⟵ Run a comparison test

5.8 Update your design.pdf

☐ **To do:** Did your final program deviate from the draft of **design.pdf** that you submitted? Update **design.pdf** to match your program.

5.9 Submit: add/commit/push/submit commit ID

You need to submit these files:

- `design.pdf`
- `Makefile`

- `nibbler.c`
- `unittests.c`
- `compress.c`
- `decompress.c`

- `nibbler.h`

☐ **To do:** `git add` all of the files above.
☐ **To do:** `git commit -m '`*reason*`'` (you can give any reason for this commit)
☐ **To do:** `git push`
☐ **To do:** Submit your final commit ID to Canvas by Wednesday November 20$^{\text{th}}$ at 11:30 pm.

6 Revisions

Version 1 Original.

References

[1] David Salomon. *Data Compression: The Complete Reference, 4th ed.*, pages 20–21. Springer-Verlag, London Ltd., 4th edition, 2007.