COMP2401 A/B/C

Assignment #3: An Efficient Camera

Bits and Bytes

Goals

In this assignment you will be writing a program in C using the Ubuntu Linux environment which is run on the course virtual machine (VM). You will be working with data representation and exploring how the same data can be represented as characters, as individual bits within those characters, and ways that we can encode those bits to be more space-efficient.

Restricted Resources

- You **may** use generative AI conversationally to learn about topics,
- You **may** use generative AI to produce small amounts of code (e.g., one-two lines)
  - Code generated by AI must be cited in comments and any conversations included as a text file alongside your submission,
  - You must be prepared to fully explain all lines generated by the AI,
  - While allowed, you are discouraged from using AI for this assignment so that you can ensure you learn the fundamentals, and are encouraged to attend TA or instructor office hours for support.
- You **must** work individually and **only use your own original work created for this term**

Learning Outcomes

- **Apply** bit operators to numeric literal values or variable contents.
- **Apply** bit mask techniques to store and retrieve bit-level information.
- **Distinguish** the bit models used for a variable defined in C

Preparation

Where to find more help for the content of this assignment:

- **Tutorial 3:** Bits and Bytes,
- **Chapter 2:** A good understanding of of Chapter 2.1-2.3 is very helpful at this stage, with emphasis on bitwise operations and data types.
- **Bits Review:** A video is posted to the Week 3 and Week 4 Weekly Notes reviewing bitwise operations, bases, and binary in C.

Assignment Context (Optional Read)

Fantastic work with the data collection program! That's going to be a big help going forward. For now, we have a new problem. We have some old black-and-white cameras wirelessly connected to our van outdoors. The problem is, it is really, really slow. I don't think the people who programmed the photo storage methods really knew much about the binary underlying their data types, and didn't make it very efficient for data transfer!

We've provided you with a simulated version of the camera that would normally send data over a network, but for now, it just provides you some sample images. We want you to print those images to the screen, then try to build some C code to reduce the amount of data stored for each photo, and then write a function to

take that output and read it in, and print it to the screen. After that we can hook your code up to the real-deal!

1. Overview

In this assignment, you have been provided with an object file and a header file: `camera.o` and `camera.h`. `camera.o` cannot be opened, as it is **not** a C source file. Instead, it contains the implementation for the `get_next_photo()` function, but it has been pre-compiled into object code. The header file `camera.h` provides the definitions needed by `camera.o`, so they will be included anywhere that `camera` functions are needed.

You will be using this `get_next_photo()` function to receive "photographs", arrays of ASCII characters, where the ASCII character `'1'` represents a black pixel and ASCII character `'0'` represents a white pixel. This function has the following prototype:

```
/*
  Pulls the next photo from our simulated camera and returns an array of
  ASCII characters representing the pixel colours.

  out: dest; An array of size at least MAX_PHOTO which will be populated
             with the characters from the simulated photograph.
  out: rows; The address of an integer to store the number of rows in
             picture that was taken.
  out: cols; The address of an integer to store the number of columns
             in the picture that was taken.

  Return: Integer representing the number of characters in the photograph
          (equivalent to rows * cols), 0 if no photo was returned
          (indicating the end of the photographs)
    - Array `dest` is populated with characters,
    - Integers `rows` and `cols` are populated with the number of
      rows and columns in the photo
*/
int get_next_photo(unsigned char dest[], int *rows, int *cols);
```

1.1. Control Flow

Watch the Sample I/O video on Brightspace for more information.

The following control flow should be handled within your `main()` function.

1. In a loop, call `get_next_photo()` until no photos remain, i.e., the size returned is 0,
2. Using your `print_ascii()` function, print the image to the screen as described in the requirements.
3. Take the array returned by `get_next_photo()` and pack the bits using your `pack_bits()` function so that it takes up about 8x less storage space in a new array of size `PACKED_PHOTO_SIZE`.
4. Call your `print_packed_bits()` function to print the photo to the screen, but this time, based on your packed bits representation.
5. Further compress the packed bits using Run-Length Encoding (RLE) in your `rle_encode()` function, storing the output in a new array.
6. Call your `print_rle()` function to print the RLE values to the screen.

2. Reading and Running the Code

First, you will need to extract and review the files in `a3-posted.tar`.

You are given a few files to get you started. You will notice that the files which rely on parts of `camera.o` will `#include "camera.h"`, which will forward declare the functions and values defined for `camera.o` so that every file can work with them consistently. `main.c` will contain your main control flow code and nothing else. `photo.c` will contain all of the code for your printing and reading of the data from `camera.o`. Your code should compile and run with a command similar to `gcc -Wall main.c photo.c camera.o -o a3`.

3. Background

This assignment is all about understanding data representation and bit manipulation, so this section gives some extra review on these topics.

3.1. ASCII and Bit Packing

It is important to recognize that `get_next_photo()` returns `characters`, specifically, characters containing **ASCII representations**. A `char` is 8 bits, or one byte. We often denote ASCII using single quotes, `' '`. ASCII is just a mapping of integer values to a table, so the decimal value `65` maps to `'A'`, `66` to `'B'` and so on, and `97` to `'a'`, `98` to `'b'`. Numbers, represented in ASCII, have a similar mapping: `'1'` is really the decimal value `49`, with `'0'` mapping to decimal `48`.

The binary to represent the **integer value** of `1` is `0b00000001`, in a single byte, and the binary to represent `0` is `0b00000000`… Naturally. But, the **character** representations map to their ASCII values, so `'1'` is `0b00110001` ad `'0'` is `0b00110000`. Clearly, taking up a whole byte to represent `0` and `1`, whether it is in ASCII representation or integer, is wasteful, as a single bit can accomplish this.

We want to perform what is called "Bit Packing". This is where we stop looking at a value as its whole value and instead look at it as a sequence of bits that can be loaded with values. For example, if `0` is a white pixel and `1` is a black pixel, `BWBWWBWW` would be `10100100`. If each of these was a single character `'1'` or `'0'`, it would take **eight bytes** to hold this value! Instead, it is better to take a single `char`, usually initialized to `0` so that it is all zeroes, and `set` each bit that represents a `1`. That way, a single character can hold up to eight bits of information.

3.2. Run-Length Encoding

Later in this assignment, you will be asked to further compress the data using a technique called "Run-Length Encoding" (RLE). Imagine you are given the following sequence of black and white pixels: `bwwwwwwwbbbbwwwwwwwwwwwwbb`. We might say that this is "1b, 7w, 4b, 11w, 2b". If we knew we always started with black, we could store this as `1,7,4,11,2`. If we used a single byte to represent each number, this would take 5 bytes.

With a regular bit-packing approach, we would only need 24 bits (3 bytes) to represent the original sequence; however, if we had very long sequences of the same colour (i.e., over 8 of the same colour in a row), we would need to use multiple bytes to represent that sequence. RLE is sometimes more efficient, sometimes less, depending on the data.

4. Instructions

For **all features** be aware of the following:

1. Print functions should **only** print the data, not modify them nor print additional details.
2. All functions should be documented with comments about their definitions which describe the purpose of the function, the parameters (including whether they are input, output, or input/output), and what the function returns. Include any important ranges or error codes.
3. You are not required to check for `NULL` pointers to simplify the assignment, but you can.
4. You will be expected to work out the details of some parts of the implementation yourself.
5. Avoid repeating code unnecessarily. Create helper functions if needed.

4.1. Printing ASCII Photos

For all features, make sure to write a separate functions to get/set/clear bits at indices as needed to improve the readability of your code.

Feature 1 Description: I want to display the photos that are currently in the ASCII format.

Feature 1 Requirements:

- **1.1.** `print_ascii()` accepts an array of ASCII characters and a number of rows and columns. This is a one-dimensional array and must be printed using the following guidelines:
  - **1.1.1.** `1` represents a black pixel and should be drawn to the screen with the `'*'` character,
  - **1.1.2.** `0` represents a white pixel and should be drawn to the screen with the `'.'` character,
  - **1.1.3.** Characters represent rows, starting with the top left moving to the right (i.e., if there are 5 columns, the sixth character will be placed on the first column of the next second line)
  - **1.1.4.** Ensure there are **no spaces** between characters,
- **1.2.** Return the appropriate error code if a character is either invalid (not a 1 or 0) or if the size is invalid,
- **1.3.** In the `main()` function, declare an unpopulated array of `unsigned char` with size `MAX_PHOTO_SIZE`, as well as two `int` values to represent the rows and columns,
- **1.4.** Call `get_next_photo()`, passing in the array of characters and the address-of (using the `&` operator) your integers and verify that you get correct output,
- **1.5.** Loop until you no longer receive a valid picture from `get_next_photo()`, i.e., when the returned size is 0.

Testing Tip: It can be helpful to create a test array to make sure your `print_ascii()` function works correctly, independently of `get_next_photo()`, such as an array like `"000010000000101000001111100010000010100000001"` which, with 9 columns and 5 rows, should produce something which resembles a capital `A`. Do **not** execute any testing code in your submission; you may wish to use separate functions for this.

4.2. Packing Bits

Feature 2 Description: I want the array of characters to be 1/8th the size by packing eight characters into a single `unsigned char` for every eight characters in the original ASCII photograph.

Feature 2 Requirements:

2.6. The `pack_bits()` function accepts the source (`src`) array, which is ASCII characters, and output to the destination (`dest`) array. The `src` is `const`, because it should not be modified by this process,

2.7. The packed bits are packed so that the first of the eight bits is stored in the **most-significant bit position**, e.g., given the `src` array `{0, 0, 0, 0, 1, 1, 1, 1}`, the first `char` of the `dest` array will be `00001111`, with zeroes in bit position `7, 6, 5, 4` and ones in bit position `3, 2, 1, 0`,

2.8. The `pack_bits()` function returns an appropriate error code if the provided size is invalid or if an unknown character was found, otherwise it returns the total number of bytes used in the packed array.

2.9. In the `main()` function after printing the ASCII version, pack the bits into a new array of size `PACKED_PHOTO_SIZE`.

Testing Tip: Consider packing a very simple array to test the output. For example, the array of characters `"010000010100001001000011"` should pack into three characters `01000001`, `01000010`, and `01000011`, which would `printf()` as characters (`%c`) as `A`, `B`, and `C`.

Testing Tip: You have been provided with a function in `camera.o` called `camera_test_packed()`. This function can be called to check if the packed bits for a given photo seem to be incorrect. It returns the number of incorrect bytes found or 0 if everything seems correct. It will also print the first few bytes that seem incorrect. Use of this function is optional, but can be helpful for debugging. It does not guarantee full marks for correctness, though.

4.3. Unpacking Bits

Feature 3 Description: I want to print the packed version of the photograph to the screen.

Feature 3 Requirements:

3.10. In `print_packed_bits()`, which accepts the bit-packed photo array, loop over each bit in each byte and print each `1` bit as a `'+'` character and each `0` bit as a `'-'` character.

3.11. Update your `main()` function to print out the packed bits version of the photograph.

4.4. Run-Length Encoding (RLE)

Feature 4 Description: I want to further compress the packed bits using Run-Length Encoding (RLE) and be able to print the RLE version of the photograph to the screen.

Feature 4 Requirements:

4.12. `rle_encode()` accepts a **packed bits array** and its size and an output array to store the RLE values.

4.13. The function cannot encode images larger than 255 rows or 255 columns, and should return an appropriate error code if the provided size is invalid.

4.14. RLE is described earlier in Section 3.2 and should be implemented as follows:

- 4.14.1. The first byte of the RLE is the row count (in pixels) of the image from 1 - 255,
- 4.14.2. The second byte of the RLE is the column count (in pixels) of the image from 1 - 255,
- 4.14.3. The third byte begins the RLE values, starting with the number of consecutive black pixels (1-bits) followed by the number of consecutive white pixels (0-bits), alternating until the entire image is represented.

4.14.4. The function should return the number of bytes used in the full array or an appropriate error code if the provided size is invalid.

4.14.5. **Note:** If there are 300 consecutive 1-bits, a byte cannot hold that value. As such, you should store `255`, followed by a `0` to indicate that there are no 0-bits, followed by `45` to indicate the remaining 45 1-bits.

4.15. `print_rle()` accepts an RLE array and its size and prints the RLE values to the screen.

4.15.1. The size can be derived from the array that is passed in

4.15.2. Each 1-bit should be printed with a `'#'` character and each 0-bit should be printed with a `' '` (space) character.

4.16. `main()` should be updated to also call `rle_encode()` on the packed bits array returned by `pack_bits()` and then call `print_rle()` to print the RLE values to the screen.

Testing Tip: Consider testing with a very simple packed array to verify your RLE implementation. For example, the packed array `{0b11111111, 0b00000000}` (16 pixels total) should produce an RLE array of `{1, 16, 8, 8}` (4 bytes total). The first byte is `1` for one row, the second byte is `16` for sixteen columns, the third byte is `8` for eight black pixels, and the fourth byte is `8` for eight white pixels.

You have also been provided with a function in `camera.o` called `camera_test_rle()`. This function can be called to check if the RLE for a given photo seems to be incorrect. It returns the number of incorrect bytes found or 0 if everything seems correct. It will also print the first few values that seem to be incorrect. Use of this function is optional, but can be helpful for debugging. It does not guarantee full marks for correctness, though.

4.5. Finishing Up

Make sure to test your code and ensure it is all working correctly. Review the video posted alongside the submission to make sure it follows the correct I/O and control flow.

For all assignments and projects going forward, you must include a **README** file and make sure your code is clean, consistent, and documented.

README: The README file can be called `README` or `README.txt` or `README.md`. The purpose of this file is to act as a kind of starting-point for your project. It should include the following:

- Your name and student number,
- A description of what the program is and does,
- Instructions for compiling and executing the program via command line,
- Any additional citations or sources when using valid external sources for information.

Make sure to submit your code according to the instructions in Section 5

4.6. Optional Side Quest Bonus

- This challenge is optional, and can be submitted to the Side Quest forum to receive marks toward your Side Quest mark,
- **Submission:** You may submit this version as your assignment, but to receive side quest marks, must also submit the files to the Side Quest forum.
- **Motivation:** Chapter 2.2 describes some very realistic uses for bitmasks; namely that data is often treated as a stream of bits and we use bitmask operations to read certain parts of the data differently from others. For example, the first few bits might represent an error code, the next few might include information about what type of data will follow. Using custom bitmasks to read this data can be very helpful!
- **Challenge:** `camera.o` contains an extra, more complicated function to take photos - `sq_next_photo()`. All of the data needed to display the image is stored within the bytes themselves, it is not all pixel data! Additionally, there are two modes; 1-bit colour mode (black-and-white) and 2-bit colour mode (4 shades). The data arrives in the following format:
- 2 bits: `01` for 1-bit, `10` for 2-bit colour mode
- 6 bits: Magnitude-only number of columns (width), minus 1
- 6 bits: Magnitude-only number of rows (height), minus 1
- Remaining Bits: Packed bits representing each pixel, depending on colour mode:
  - In 1-bit mode, each bit is a pixel, with `1` being printed with `#` and `0` being printed with `.`
  - In 2-bit mode, each pair of bits is a pixel, with the following prints:
    - `00`: `'.'`, `01`: `':'`, `10`: `'*'`, `11`: `'@'`
- The return code of the function will be the number of bytes used for the image, and `0` if there is no remaining image data.
- Write your own print function to print out the bits, either in 2-bit or 1-bit mode as specified in the bits. A sample prototype is available in `photo.h`, but you can write your own.

Note: This can be challenging! The sizes are one less than the actual sizes because this allows us to say, for example, we have a 64x64 sizes image by using all size bits `0b111111`, which is normally 63. We know the image will always have at least 1 row and 1 column, so this helps us preserve a bit.

Tips: When shifting, consider the number of bits that you are moving and how that affects how many bits you need to shift. Remember that the first pixel colour does not start on Byte 1, or at the most significant bit of Byte 2. Remember to add one to the number of rows and columns!

5. Packaging and Submission

For this assignment, you will submit a single archive file, saved as a `.tar` compressed in Linux using a command similar to `tar -cvf a3.tar <file1> <file2>`, which includes **all files needed to run your code** and your README file, which have been updated to include the necessary information. It **must** have the `.tar` file extension and be a valid tar file.

6. Grading

Grading information is a general reference, and small changes might be made as unexpected cases necessitate, but will be visible on the assignment rubric provided as feedback.

The autograder **does not** use your `main.c` file, instead grading each function individually against a series of tests, and will provide some general information about the tests that fail in the feedback. If we can not compile using our own `main.c` file, you will receive a **zero** for the automated tests. Changes to described function names, data types, etc. can all lead to the autograder failing.

For TA review, the TA will compile and execute your full code (i.e., run your `main.c` file) according to the instructions in your README file and attempt a basic execution to evaluate the main control flow, error message quality, and print quality.

After receiving feedback, you will have **one week** to revise and re-submit your code to the revised submission portal on Brightspace. You will then have one week from receiving that grade to dispute your mark. Review the course outline for information on grade dispute and second chance submission policies.

6.1. Automated Grading, Per-Function Execution

No marks are awarded if the autograder cannot untar your work or the code cannot compile. Test your submission early to avoid issues at the deadline. Always download your submission and test it in an empty directory before the deadline. A full and up-to-date rubric will be provided with your feedback. Some samples of the tests that will be performed have been included in the provided object file and you are encouraged to consider writing tests like these before starting future assignments.

- `print_ascii()` correctly prints test data
- `pack_bits()` correctly packs the bits of test data and returns the correct values
- `print_packed_bits()` correctly prints out the packed bits of test data
- `rle_encode()` correctly encodes test data and returns the correct values
- `print_rle()` correctly prints out the RLE

6.2. Manual Review

The manual review will be based on a simple rubric which observes the following:

- **Execution:** Main execution control flow works correctly as specified.
- **Packaging:** All required files are included and contain the correct information.
- **Design:** Code is well-structured, with consistent naming and no unnecessary duplication.
- **Documentation:** Functions contain documentation comments above their definitions, detailing their purpose, parameters, and return values. Must include whether parameters are input, output, or input/output (read, write, or read+write).
- **Approach:** Algorithms and conventions required from the submission are followed.
- Deductions may be applied for, at minimum:
  - Using global variables (variables defined outside of a function)
  - Using techniques that are not described in the class or course notes which bypass the assignment requirements

Appendix A: Code Style guide

We follow good design principles to be consistent, make it easier to read and understand our work, to make it easier to modify and update our work, and to simplify debugging.

```c
int main(void) {

}

// or

int main(void)
{

}
```

Use a consistent brace style.

```c
// Print each odd value in the array
for (i=0; i<size; i++) {
  if (arr[i] % 2 != 0) {
    printf("%d", arr[i]);
  }
}
```

Use comments to explain behaviour that isn't obvious at-a-glance.

```c
int i; // Not good

int main(void) {
  for (i=0; i<42; i++) {
    // ...
  }
}
```

Do not use global variables (like `i`) unless instructed to.

```c
// Bad example...
for (i=0; i<MAX_ELEMENTS; i++)
  if (i % 2 == 0)
    printf("At least I indented...");
  if (i % 3 == 0) printf("A lot wrong here.");

```

Generally avoid loops and conditions without parentheses (It can be tough to debug!)

```c
/*
    Removes the id and reading for any readings strictly under `minimum` from the `ids`
and `readings` arrays.

    in/out ids[]:       Array of IDs associated with EMF readings in `readings`
    in/out readings[]:  Array of EMF readings to check against the minimum
    in     size:        Number of elements in `ids` and `readings`
    Returns:
        - Updated number of elements in the arrays
*/
int sort_entries(int ids[], float readings[], int size) {
```

Include function documentation in comments which explains the purpose of the function, what the parameters do, and what it can return.

- Input Parameters: The data will be read by the function.
- Output Parameters: The data will be modified by the function.
- Input/Output Parameters: The data will be both read and modified.

Appendix B: Example README

README files are very common for code projects. They typically include similar information. On website like GitHub, **Markdown** is commonly used to format these files. Markdown is a format to make text files easily rendered with styles while still being readable. It is not strictly necessary to use Markdown, but it is a helpful tool to learn! There are often small variations in the syntax, but generally these are good to use:

```
# This is a Header

## This is a Second-Level Header, like "Section 2.1"

*This text is italic*, **This text is bold**,
_This is usually italic as well_. This is just regular text.

[This is a Link](https://brightspace.carleton.ca)

This `is a line of code` inside a paragraph.

* This is
* A bullet-point list

```c
int main(void) {
  // This is a block of code with a language specified
}
```
```

For this course, you can format your README however you like, but make sure to include your Student ID, your name, a description of the program, any citations required, and instructions for compiling and running the assignment. Consider something like this (but with proper details of course):

```
# Assignment 1, Data Entry
This program has a few features and they do things that a student would describe more specifically that this vague sentence is. Maybe I even take a moment to remind myself of what the learning goals were so when I read this code in 5 years I understand where I was!

## Building and Running
1. Open a terminal and navigate to the folder containing the program's files,
2. Type the command `whatever compiles this`,
3. You should see something specific when that happens to know it worked!
4. From that directory, you can run `this command` to execute the project.
5. If you want to test it, you can use `this other command`!
6. Maybe some Usage instructions to teach someone how to interact with your program once it is running

## Credits
- Developed individually by Me (133700042)
- Performed Q&A with GPT-4 via ChatGPT (September 37th version) to learn about `scanf()`, transcript of chat attached as `CHATGPT-SCANF.txt`

```