CSC 116
Fundamentals of Programming with Engineering Applications II

Note 1 **This assignment is to be done individually**
Note 2 You can discuss the assignment with others, but copying code is prohibited.

A note on Academic Integrity and Plagiarim

Please review the following documents:

- Standards for Professional Behaviour, Faculty of Engineering:
  `https://www.uvic.ca/engineering/assets/docs/professional-behaviour.pdf`
- Policies Academic Integrity, UVic:
  `https://www.uvic.ca/students/academics/academic-integrity/`
- Uvic's Calendar section on Plagirism:
  `https://www.uvic.ca/calendar/undergrad/index.php#/policy/Sk_0xsM_V`

Note specifically:

> Plagiarism
> Single or multiple instances of inadequate attribution of sources should result in a failing grade for the work. A largely or fully plagiarized piece of work should result in a grade of F for the course.

A program you submit will be considered a **piece of work**. You are responsible for your own submission, but you could also be responsible if somebody plagiarizes your submission.

1 Objectives

After completing this assignment, you will have experience with:

- File I/O in binary mode
- IO manipulators

2 Introduction

You think it will never happen to you: you are reviewing the photos in your digital camera, and suddenly, unexpectedly, and stupidly, you format your memory card. Yes, it has happened to me too[1]. The last time, I had just taken around 100 images and wanted to erase one of them. Due to my lack of attention (something that seems common) I pressed "Format/OK".

Fortunately we have tools that recover photos from a formatted memory card. Surprisingly, the "technique" behind these tools is simple, particularly when you have not deleted any photos since the last format of the card.

The details about how files are stored in a card are complex, but for our purpose, suffice to say that any memory card is a big file, as big as bytes there are in a card. For example, an 8 Megabyte card has 7979008 (yes, we are cheated, they should be 1024 * 1024 * 8 bytes, but that is a different story). When the camera formats this card some sections of it are reserved for information such as directories. The

1. See `http://photo.net/digital-darkroom-forum/00BDUc`.

minimum amount of information that can be allocated in a card is 512 bytes (a sector). Hence, we can see a memory card as a long sequence of sectors, each of 512 bytes.

When the camera saves a photo it will ask the card for empty space for it (as much as needed for the photo). In an ideal world, the photo is stored in contiguous sectors. Fortunately, this is true when you format your card and start using it without ever erasing a photo. In that case all your photos are nicely packed one after another.

However, if you have deleted one or more and taken more photos, the photos are split into many different areas of the memory card and their recovery becomes significantly more difficult (but not impossible).

What are the use cases for our program? Two:

- You took some photos and by mistake reformatted your memory card.
- Your deleted by mistake the wrong photo (and you have not taken any other photos).

In the future, if you ever do this, you can quickly go home, and undust your assignment from csc116, and save the day. At least that is the original goal.

How? That is where knowledge becomes power.

Most files use the first bytes to store what is known as its magic number. By reading these bytes you can tell what type of file it is. The magic number of JPEGs is the two bytes `0xff 0xd8` (remember, in C/C++ notation `0xAA` means byte of value AA in hexadecimal). Because a JPEG **always** starts at the beginning of a sector we can find a *potential* image by looking at the first 2 bytes of every sector, from the beginning to the end of the memory card data.

So we start from the first sector, check the first 2 bytes, if they are a JPEG, great, we process it. Otherwise we jump to the beginning of the next sector... and so on, until we run out of sectors.

What happens when we detect a *potential* JPEG? We must follow its trail. JPEGs are surprisingly simple in their internal organization. As the Wikipedia explains, a JPEG image consists of a sequence of records. Each record has the following data:

- it begins with a record-id (2 bytes)
- an optional *length* (2 bytes), and
- an optional "payload".

The *length* corresponds to the number of bytes in the payload + 2 (the number of bytes used by the *length* itself). See the Wikipedia for more details `http://en.wikipedia.org/wiki/JPEG` under Syntax and Structure. If you look up the magic number described above (`0xff 0xd8`) in this Wikipedia page, you will see that it presents a "Start of Image" record (SOI).

Most records contain a *length* field (the two important exceptions are the first and last records, *Start of Image* and *End of Image*, respectively). This length is stored in two bytes. You might know this already: number are stored differently in different architectures. Hence, the two bytes in the JPEG are stored in "network order" (also known as big-endian, see Wikipedia `http://en.wikipedia.org/wiki/Endianness`). You must convert those two bytes to a valid integer using a function we have provided `big_endian_2_bytes_to_int`.

A JPEG is therefore composed of 3 parts, as exemplified in figure 1.

- A sequence of records, the first is always `0xFF 0xD8` and the last is always `0xFF 0xDA`.
- The image data. This is a sequence of bytes.
- An *End of Image* record: `0xFF 0xD9`. This sequence of two bytes will never exist in the image data.

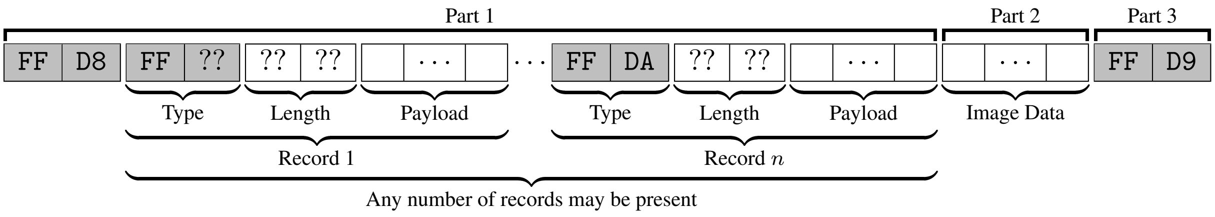

Figure 1: JPEG internal format

To help you understand this format, we recommend you start by implementing the scanner part of the assignment (see below) that can help you visualize the internal structure of a known JPEG file. For example, for the file tests/test-01.jpg, these are the records, their lengths and offsets, including the data area. Notice how the last record is located exactly at the last two bytes of the file.

```
Record ffd8 of length        0 at offset        0
Record ffe1 of length    16432 at offset        2
Record ffc4 of length      418 at offset    16436
Record ffdb of length      132 at offset    16856
Record ffc0 of length       17 at offset    16990
Record ffda of length       12 at offset    17009
Data starts at byte offset 17023
Record ffd9 of length        0 at offset   254786
File contains 254788 bytes
Processed 1 photo(s).
```

The format of each record printed is the following:

- The string *Record* followed by one space.
- The hex values of the next two bytes (eg. ffd8) followed by one space.
- The string *of length* followed by one space.
- The size of the record, aligned to the right with a width of 8 spaces, followed by one space.
- The string *at offset* followed by one space.
- The offset of the record, aligned to the right with a width of 8 spaces.

Given the binary dump of a memory card, we will start by scanning the beginning of each sector (every 512 bytes). Call this sector the i-th sector. If the sector does not start with the magic number of the JPEG we skip it. If the sector starts with the magic number of the JPEG, we start reading the file as if it was a JPEG until something goes wrong (for example, a record does not start with `0xFF`) until we find the *End of Image Record*.

If something goes wrong (i.e. we do not find a valid record, or we reach the end of file before the End-of-Image record), we backtrack and read the i+1 sector.

If we find the *End of Image Record*, we save the bytes from the *Begin of Image Record* (start of i-th record) to *End of Image Record* (both included) to a file (we have recovered a JPEG!). The next sector to scan will be after the sector that contains the *End of Image Record* where we will try to find another image.

We keep doing this until we have scanned all sectors. Hence, our program finishes when all the sectors have been traversed.

Your task, should you choose to accept it

Your are required to write the program `recover` that scans and (optionally) recovers the JPEGs from a formatted memory card file.

1. It should take two arguments, the name of the file to scan and the action to execute: *scan* or *recover*
2. In scan mode, the program prints the records in the JPEG, their lengths and offsets (see above).
3. In recover mode, the program prints the same information as above, plus create the corresponding jpegs found in the file. It should save each image with the names `recovered-0000.jpg`, `recovered-0001.jpg`, `recovered-0002.jpg`, etc. in the current directory. Every recovered file should be a valid JPEG.

In both cases the program should output the number of photos recovered. See example above.

An algorithm to recover JPGs from a file system dump

This is the very general algorithm needed to recover photos. It assumes that all blocks that look like a JPG are contiguous. You can read the entire file in memory and process it in memory, or read data as required. It assumes that blocks are 512 bytes.

```
currentOffset <- 0
while(there is still input) {
   skip = 512
   inspect 2 bytes at currentOffset
   if these two bytes are the JPEG magic number (0xFF 0xD8):
      process photo at currentOffset
      if the photo is valid:
         output photo
         skip = skip * number of blocks that the photo occupied
   currentOffset += skip
```

To process a photo (if at any point you reach the end of the file, the image was invalid):

```
// assume that currentOffset contains the beginning of the photo

offset <- currentOffset
// skip records until finding the record 0xFF 0xDA (Start of Scan)
while (record at offset is not 0xFF 0xDA)
   offset += length of record at offset

// skip image data -- until reaching the End-of-Image record
while (record at offset is not 0xFF 0xD9)
   offset++

verify that record at offset is 0xFF 0xD9
// add to the offset the 2 bytes of this record
offset += 2

// offset now contains the end of the photo
// compute its length
length of photo <- offset - currentOffset

// the photo starts at currentOffset and ends at offset-1
photo <- data from currentOffset to (offset - 1)
```

Additional Hints

- You can read any number of bytes into a std::string. There are many ways to do this. This is one example:
```
// input is an std::ifstream
    std::string st {};
    st.resize(lengthToRead); // resize string to the desired length
    input.read(&st[0], lengthToRead); //read data into string
```
  You can then process the string *st* as a sequence of bytes of length *st.size()*. Another alternative is to use the get() method.
- Do not forget to convert the length of the record from Big-endian to Little-endian using `big_endian_2_bytes_to_int`. See `recover-photos.cpp` for its declaration.
- You can compare the record types against strings by creating 2 character strings that contain the given record. For example, if the string `st` contains a string of length 2. You can do:
```
if (st == "\xff\xd9")
```
  Note the use `\xAB` to denote a character with hexadecimal value AB.
- Alternatively, you can convert a two byte string to an integer and compare the result to a 16-bit hexadecimal value. If the string `st` contains a string of length 2, you can write
```
if (big_endian_2_bytes_to_int(st) == 0xffd9)
```
- Remember to set your output stream (for each recovered photo) to binary.

This program is not complex nor long, but requires you to be careful. Your instructor's solution is less than 160 lines of code.

3 Code provided

Modify the file `recover-photos.cpp`. Implement the function `Recover_Photos`. Its parameters are the file name to open and the operation to execute.

3.1 Further notes

- Your program should be all contained in a single source code file `recover-photos.cpp`.
- The input file can be of any size.
- No JPEG will be larger than 25 megabytes.
- If you run `recover_photos` on a JPEG, the output should be the same as the input JPEG. This is because the JPEG starts at the beginning of a sector offset.

4 Tests provided

Test 01 to 05 do scans only. Test 01 to 03 are JPEGs and 04 and 05 are dumps of 2 memory cards. Test 04 contains 6 photos and Test 05 20.

Test 06 to 10 exercise the recover functionality. They use the same inputs (in the same order) as the first five tests. The expected images are located in `tests/rec-06` to `tests/rec-10`.

5 Evaluation

Solutions should be:

1. Correct. They should pass all the tests we have provided.
2. In good style, including indentation and line breaks

What to submit

- Using conneX, submit your version of the file `recover-photos.cpp`. **Do not submit any other files or alter the filename of `recover-photos.cpp`**.
- You should assume that your submission was received correctly only if both of the following two conditions hold.
  - You receive a confirmation email from conneX.
  - You can download and view your submission, and it contains the correct data.
- In the event that your submission is not received by the marker, you will need to demonstrate that both of the above conditions were met if you want to argue that there was a submission error.