**Griffin Hurt**
Undergraduate Teaching Fellow
griffhurt@pitt.edu
https://griffinhurt.com

Spring 2024, Term 2244
Friday 2 PM Recitation
Feb 16th, 2024

Slides adapted from
Shinwoo Kim, Martha Dixon, and Vinicius Petrucci

Department of Computer Science
School of Computing & Information
University of Pittsburgh

# Recitation 5: File I/O and Project 1

- Agenda
- Course News
- File I/O
- Quiz!
- Project 1

# Agenda

File I/O in C
- Standard integer sizes
- Reading/writing files

Quiz!

Project 1

# Basics of File I/O

## Reading and writing files in C

```
[ ~ ]$ hexdump -C binary_file_example
00000000  41 00 41 00 00 00 42 00  42 00 00 00 43 00 43 00  |A.A...B.B...C.C.|
00000010  00 00 44 00 44 00 00 00  45 00 45 00 00 00 46 00  |..D.D...E.E...F.|
00000020  46 00 00 00 47 00 47 00  00 00 48 00 48 00 00 00  |F...G.G...H.H...|
00000030  49 00 49 00 00 00 4a 00  4a 00 00 00 4b 00 4b 00  |I.I...J.J...K.K.|
00000040  00 00 4c 00 4c 00 00 00  4d 00 4d 00 00 00 4e 00  |..L.L...M.M...N.|
00000050  4e 00 00 00 4f 00 4f 00  00 00 50 00 50 00 00 00  |N...O.O...P.P...|
00000060  51 00 51 00 00 00 52 00  52 00 00 00 53 00 53 00  |Q.Q...R.R...S.S.|
00000070  00 00 54 00 54 00 00 00  55 00 55 00 00 00 56 00  |..T.T...U.U...V.|
00000080  56 00 00 00 57 00 57 00  00 00 58 00 58 00 00 00  |V...W.W...X.X...|
00000090  59 00 59 00 00 00 5a 00  5a 00 00 00              |Y.Y...Z.Z...|
0000009c
```

# What we have seen so far …

- **In lab 0, you (maybe unknowingly) used command line arguments to interact with your program**
  - When you ran `./calculator 4 5 +`
- **In lab 1, you used the standard I/O stream(s)**
  - `printf()`, `scanf()`, and other `<stdio.h>` functions
- **This week, we'll learn to read and write from files on your computer**
  - which you will need to do for the first project

# What is a file?

- **In C, a file is simply a sequence (*stream*) of bytes:**
  - Text files (or ASCII file) is sequence of ASCII code, i.e., each byte is the 8 bit code of a character (*.txt, *.c, etc.)
  - Binary files contains the original binary number as stored in memory (*.pdf, *.doc, *.jpg, etc.)

```
0000000 0000 0001 0001 1010 0010 0001 0004 0128
0000010 0000 0016 0000 0028 0000 0010 0000 0020
0000020 0000 0001 0004 0000 0000 0000 0000 0000
0000030 0000 0000 0000 0010 0000 0000 0000 0204
0000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
0000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc
0000060 00fc 1819 0019 9898 0098 d9d8 00d8 5857
0000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
0000080 8888 8888 8888 8888 288e be88 8888 8888
0000090 3b83 5788 8888 8888 7667 778e 8828 8888
00000a0 d61f 7abd 8818 8888 467c 585f 8814 8188
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
00000c0 8a18 880c e841 c988 b328 6871 688e 958b
00000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
00000e0 3d86 dcb8 5cbb 8888 8888 8888 8888 8888
00000f0 8888 8888 8888 8888 8888 8888 8888 0000
0000100 0000 0000 0000 0000 0000 0000 0000 0000
```

A hex dump of the 318 byte Wikipedia favicon

# Opening files with fopen()

```
FILE *fopen(const char * pathname, const char*mode);
```

```
> FILE* pt = fopen("E:\\PATH\program.txt","w");
```

- opens the file whose name is the string pointed to by pathname and associates a stream with it.
- returns a pointer (of type FILE) to the stream

# Opening Files with `fopen()`

```
*fopen(const char * filename, const char * mode );
```

**Modes:**

- `r`: opens an existing file for reading.
- `w`: opens a file for writing.
  - If `filename` does not exist, new file is created.
  - starts writing at the beginning of file.
- `a`: opens a text file for writing in appending mode.
  - If `filename` does not exist, new file is created.
  - start appending content in the existing file content.
- `r+`: opens a file for both reading and writing.
- `b`: indicates file is a binary file
- and more…
  - Use **`man fopen`** to learn more

# `fread()` lets us read, `fwrite()` lets us write

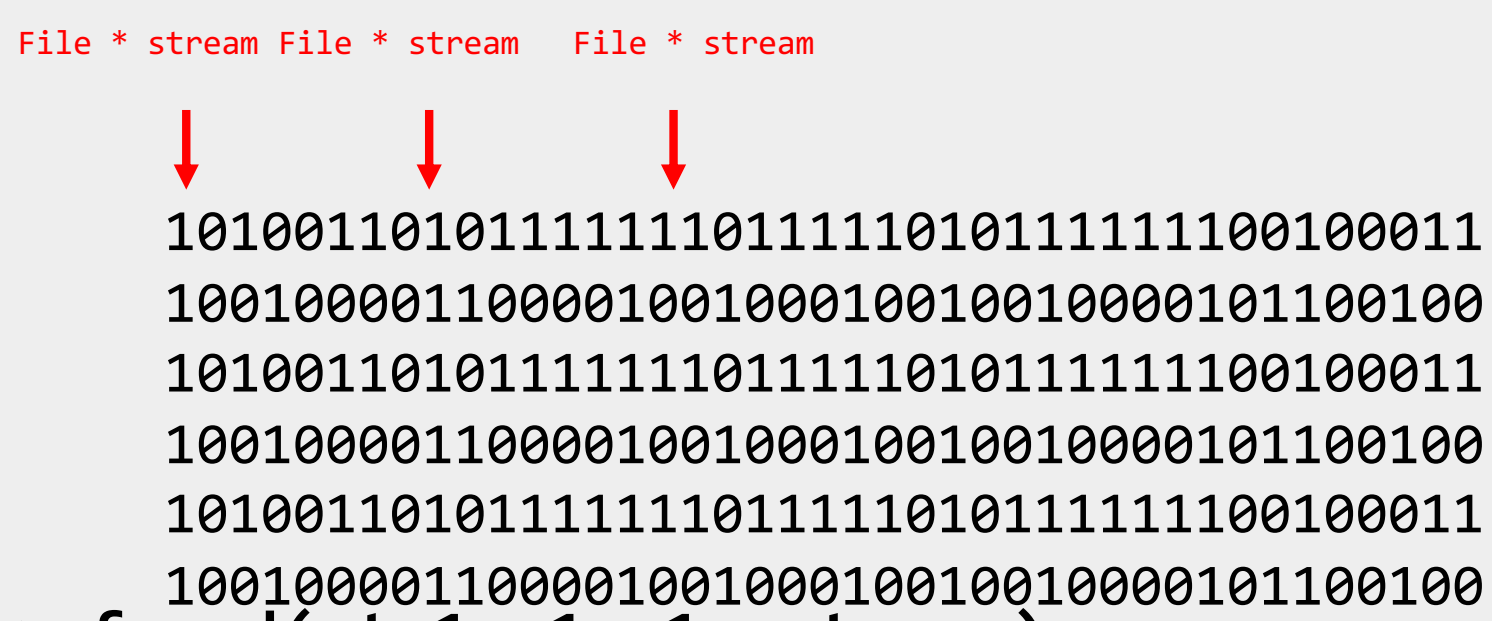
```
fread(void *ptr, size_t size, size_t nmemb, FILE* stream);
```

- reads `nmemb` items of data each `size` bytes long
- from `stream`
- stores them at the location given by `ptr`.

```
fwrite(const void *ptr, size_t size, size_t nmemb, FILE * stream);
```

- writes `nmemb` items of data each `size` bytes
- to the `stream`
- from the location given by `ptr`.

# Reading and writing moves the pointer




```
> fread(ptr1, 1, 1, stream)
> fwrite(ptr1, 1, 1, stream)
```

# Example

```
> fread(ptr1, 1, 1, stream)
```

This reads 1 byte and moves the file position indicator by 1 byte (8 bits).

```
> fread(ptr1, 4, 1, stream)
```

This reads 1 block of 4 bytes, moving the file position indicator by 4 bytes (4 * 8 = 32 bits).

```
> fread(ptr1, 4, 2, stream)
```

This reads 2 blocks of 4 bytes each from the file stream, moving the file position indicator by $4 \times 2 = 8$ bytes (8 * 8 = 64 bits).

# We can rewind or fast-forward with `fseek()`

```
fseek(FILE *stream, long offset, int whence);
```

- sets the file position indicator for the stream
- new position (measured in bytes) = `offset` + `whence`.

**`whence`:**

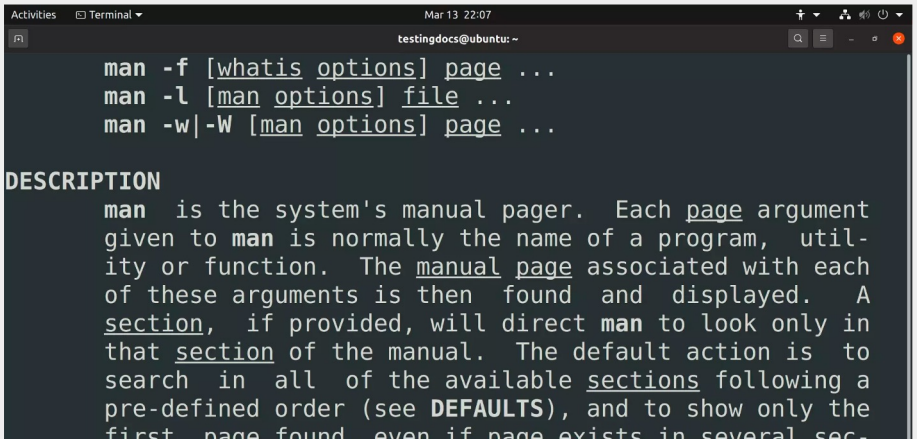
- `SEEK_SET` - from start-of-file
- `SEEK_CUR` - from current position
- `SEEK_END` - from end-of-file

# Example

- **fseek(file, 10, SEEK_SET)**
  moves the file position indicator 10 bytes from the beginning of the file.
- **fseek(file, 10, SEEK_CUR)**
  moves the file position indicator 10 bytes forward from the current position in the specified file stream.
- **fseek(file, 10, SEEK_END)**
  moves the file position indicator 10 bytes before the end of the specified file stream.

# Always remember to save (and close) your files!

- **Just like memory leaks, you may also get file handle leaks**
  - If you use `fopen()`, always remember to `fclose()`
    - `int fclose(FILE* filePointer)`
      - returns `0` on success!
- **If you are confused about these functions → Consult the `MANual`**




Thoth `man` errors: *try* `MANPATH= man 3 fopen`

# Quiz! (for participation again)

Password: _____

# Project 1

Quick Guide

# Project Brief

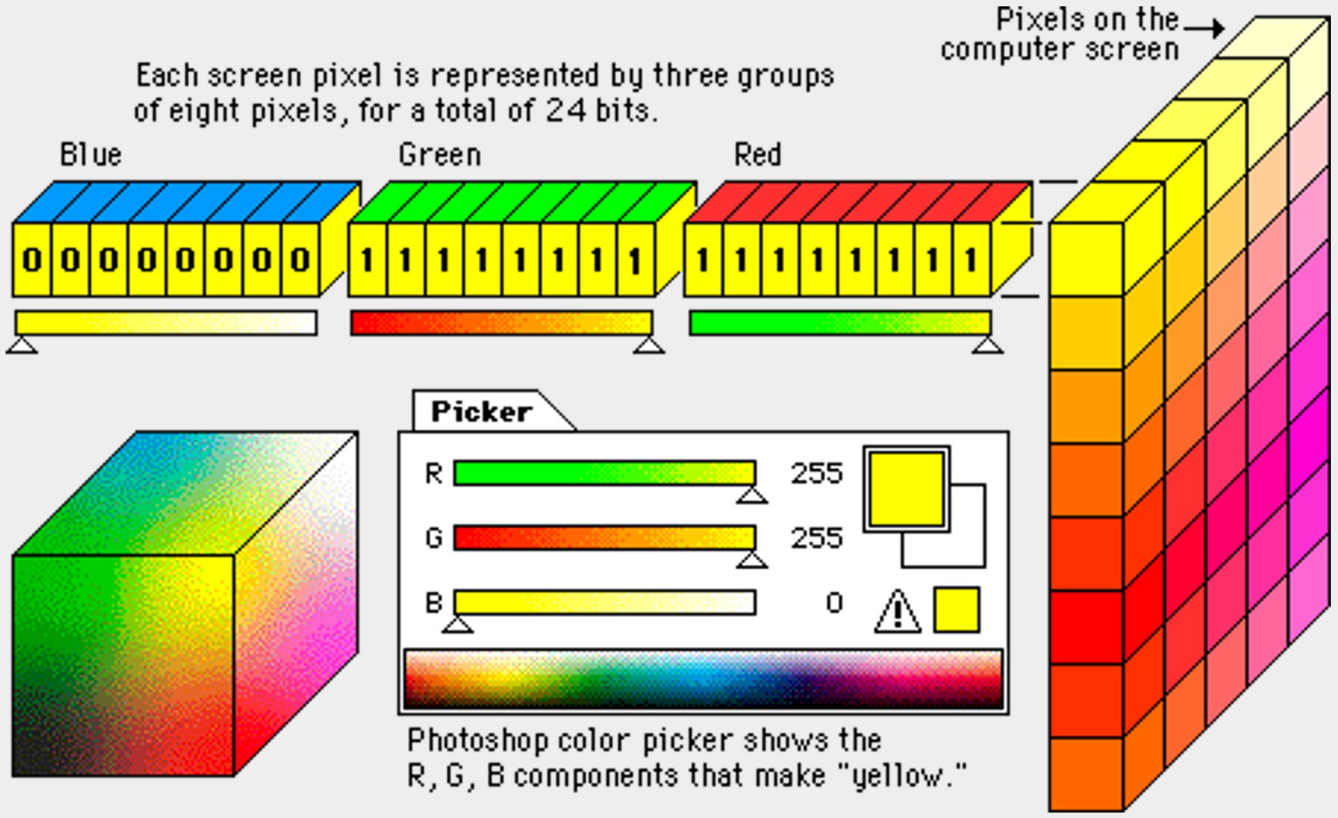
The goal of this project is to hide another image in a BMP file

- Steganography is the process of hiding data in an image
- BMP is a standard image format

`*.BMP` ⇒ Bitmap Image File

- Container format for a big array of pixels (picture cells)
- Each pixel is represented by a 24-bit number:
  - 8 bit for Red (0-255)
  - 8 bit for Green (0-255)
  - 8 bit for Blue (0-255)

# Pixels


Pixels on the computer screen
Each screen pixel is represented by three groups of eight pixels, for a total of 24 bits.
Blue
Green
Red
0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
Picker
R
255
G
255
B
0
Photoshop color picker shows the R, G, B components that make "yellow."

# Step 1. Read the BMP file

Your task is to read the BMP file and print its header to the screen

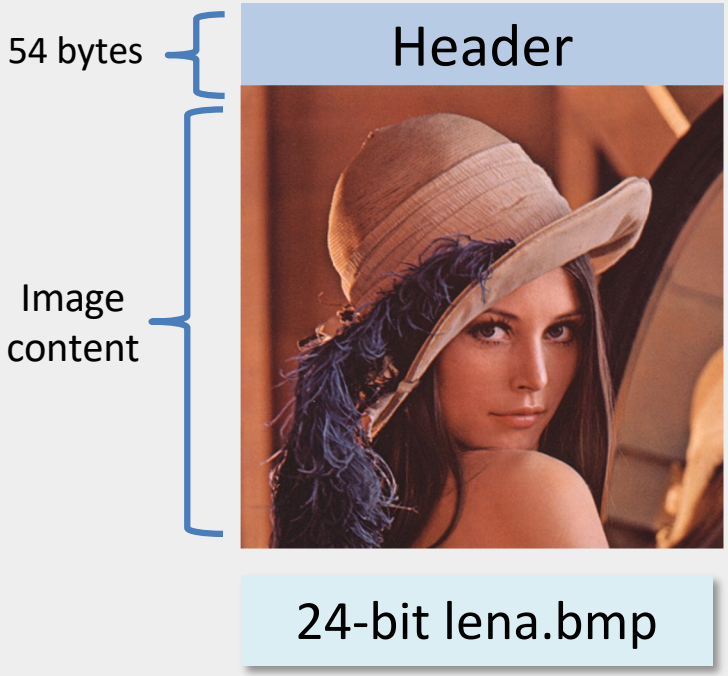
Hint: defines `struct`s and read the structs using `fread(&stuct,...)`

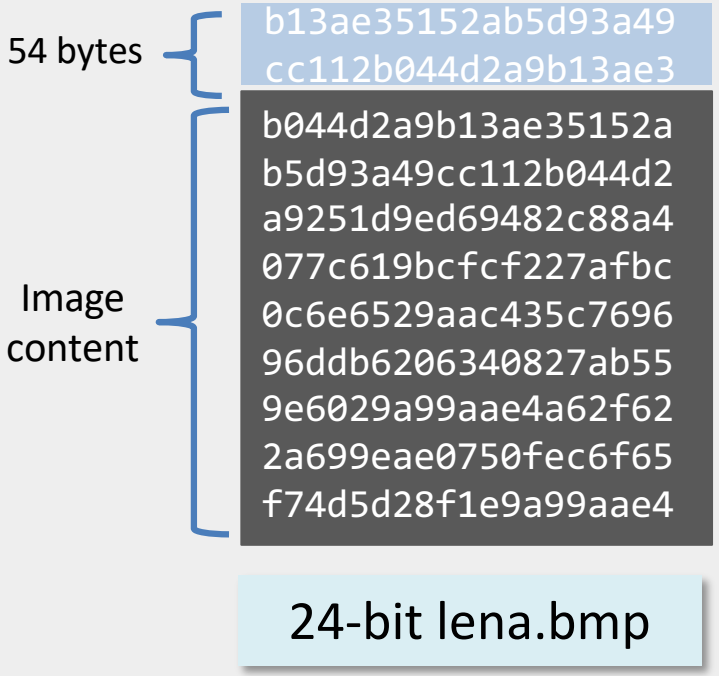
IMPORTANT NOTE: use `fopen(<filename>, “rb+”)` to mitigate text loading issues

```
$ ./bmp_steganography --info supported_bmp_file.bmp
=== BMP Header ===
Type: BM Size: 2073654
Reserved 1: 0
Reserved 2: 0
Image offset: 54
=== DIB Header ===
Size: 40
Width: 960
Height: 720
# color planes: 1
# bits per pixel: 24
Compression scheme: 0
Image size: 2073600
Horizontal resolution: 7559
Vertical resolution: 7559
# colors in palette: 0
# important colors: 0
```

# BMP File

The beginning of the BMP is a header which contains metadata (key details about the picture)



# BMP Header

Header

File header (14 bytes)

DIB Header (40 bytes)

| *Bitmap File Header* | |
|---|---|
| Identifier (ID) | 2 |
| File Size | 4 |
| Reserved | 4 |
| Bitmap Data Offset | 4 |
| *DIB Header* | |
| Bitmap Header Size | 4 |
| Width | 4 |
| Height | 4 |
| Planes | 2 |
| Bits Per Pixel | 2 |
| Compression | 4 |
| Bitmap Data Size | 4 |
| H-Resolution | 4 |
| V-Resolution | 4 |
| Used Colors | 4 |
| Important Colors | 4 |

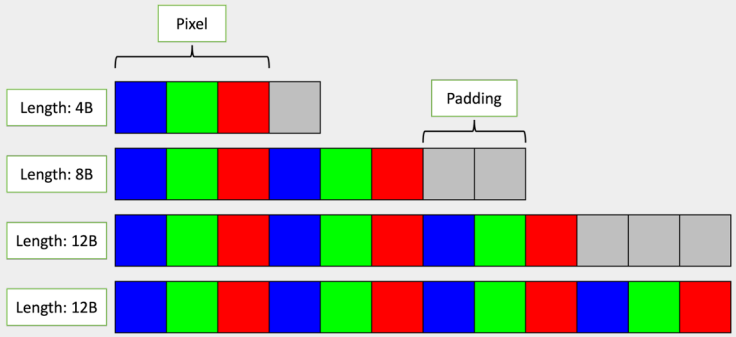
# Size of BMP

## Size of BMP file

- Size of Header + Size of Image
  - Size of Image = Width * Height * Size of Pixel (3-bytes)

## Note. Width must account for padding

- Padding is applied if length of each row is not a multiple of 4 Bytes
- Basic formula is `4 - ((width*3) % 4)` BUT special case for 0…



# Example BMP Header

```
$ ./bmp_steganography --info supported_bmp_file.bmp
=== BMP Header ===
Type: BM
Size: 2073654
Reserved 1: 0
Reserved 2: 0
Image offset: 54

=== DIB Header ===
Size: 40
Width: 960
Height: 720
# color planes: 1
# bits per pixel: 24
Compression scheme: 0
Image size: 0
Horizontal resolution: 0
Vertical resolution: 0
# colors in palette: 0
# important colors: 0
```

First two bytes must be BM (not Nul-terminated)

Keep reserved values as zero

See handout for the rest.

# Phase 2: Swap the nybbles

Use bitwise shifts and masking to move the least significant bits (last 4 bits) to the most significant bits (top 4 bits)

- Might not need to use a mask depending on the implementation

For ”hide”, use the nybble from the other BMP file instead of just swapping the ones in the first file

- You’ll need to read in pixels from both files

# Phase 3. Rewrite the BMP

- `fseek()` to get back to the beginning of pixels (use header offset)
- `fwrite()` to the file

## Caveats

- Pixels are `BGR`; Pixels are stored directly in the image section
- Each row has padding
- Pixels are stored Bottom → Top
  - Shouldn't matter if you read to a pixel array

# Remarks

See handout for

- Reading command line arguments
- **Compactness of Structs**
  - !!!
- Makefiles