pset5: Forensics

Tommy MacWilliam

tmacwilliam@cs50.net

October 16, 2011

Today's Music

- Kap Slap
  - E.T. Feel Starry Eyed
  - Remember the Collapse
  - Till Silvia Saves the World
  - All of the Nights

Today

- file I/O
- bitmaps
- `copy`
- `whodunit`
- `resize`
- `recover`

File I/O

- files are just a sequence of bytes
- input: reading those bytes from a file
- output: writing some bytes to a file

File Position Indicator

- each file has an associated file position indicator: where you are in the file
  - reading/writing bytes will start from the current position of the file position indicator
  - after reading/writing bytes, file position indicator will move forward

Opening Files

- `FILE *inptr = fopen(“clue.bmp”, “r”);`
  - open `clue.bmp` for **reading**
- `FILE *outptr = fopen(“verdict.bmp”, “w”);`
  - open `verdict.bmp` for **writing**

Reading Files

- `fread(&data, size, number, inptr);`
  - `&data`: pointer to a `struct`, which will contain bytes of file once `fread` finishes
  - `size`: size of each element to read
  - `number`: number of elements to read
  - `inptr`: `FILE *` to read from
- `fread(&data, sizeof(RGBTRIPLE), 2, inptr)` and `fread(&data, 2 * sizeof(RGBTRIPLE), 1, inptr)` are equivalent

Writing Files

- `fwrite(&data, size, number, outptr);`
  - `&data`, `size`, `number`: same as before!
  - `outptr`: `FILE *` pointer to write to
- `fputc(data, outptr);`
  - `data`: `char` to write to the `FILE *` specified by `outptr`

Seeking Files

- `fseek(inptr, amount, from)`
  - `inptr`: `FILE*` to seek in
  - `amount`: number of bytes to move cursor
  - `from`:
    - `SEEK_SET` (beginning of file)
    - `SEEK_END` (end of file)
    - `SEEK_CUR` (current position in file)

File I/O and Structs

- example time!
  - `io.c`

Bitmaps

- just like any file, a bitmap is just an arrangement of bytes
- each color represented by 3 bytes (aka scales from 0-255)
  - amount of blue
  - amount of green
  - amount of red

Bitmap Colors

- `0000ff`: no blue, no green, lots of red
  - aka red
- `00ffff`: no blue, lots of green, lots of red
  - aka yellow
- `3c14dc`: some blue, a little green, and a lot of red
  - aka crimson

RGB Triples

- pixels are represented by `RGBTRIPLE` `structs`

```
// create a red triple
RGBTRIPLE triple;
triple.rgbtBlue = 0x00;
triple.rgbtGreen = 0x00;
triple.rgbtRed = 0xff;
```

Padding

- size of each scanline must be a multiple of 4 bytes (recall each pixel is 3 bytes)
- if number of pixels per line $\times$ 3 is not a multiple of 4, we need padding
  - where padding is just `0`s to make the number of bytes be a multiple of 4

Padding Examples

- $padding = (4 - (width \times sizeof(BYTE)) \,\%\, 4) \,\%\, 4$

width	sizeof(BYTE)	padding
1	3	1
2	3	2
3	3	3
4	3	0
5	3	1

Header

offset	type	name	
0	WORD	bfType	BITMAPFILEHEADER
2	DWORD	bfSize	
6	WORD	bfReserved1	
8	WORD	bfReserved2	
10	DWORD	bfOffBits	
14	DWORD	biSize	BITMAPINFOHEADER
18	LONG	biWidth	
22	LONG	biHeight	
26	WORD	biPlanes	
28	WORD	biBitCount	
30	DWORD	biCompression	
34	DWORD	biSizeImage	
38	LONG	biXPelsPerMeter	
42	LONG	biYPelsPerMeter	
46	DWORD	biClrUsed	
50	DWORD	biClrImportant	
54	BYTE	rgbtBlue	RGBTRIPLE
55	BYTE	rgbtGreen	
56	BYTE	rgbtRed	

Header

- `biSizeImage`: total size of image (in bytes), including pixels and padding
- `biWidth`: width of image (in pixels), not including padding
- `biHeight`: height of image (in pixels)

Header

- `BITMAPFILEHEADER` and `BITMAPINFOHEADER` are `structs` defined in `bmp.h`
  - create using `BITMAPFILEHEADER bf;`
  - read data into the struct with `fread(&bf, ...);`

xxd

- `xxd -c 24 -g 3 -s 54 smiley.bmp`
  - display 3 bytes at a time, starting from the 54th byte, in 8 columns per line

copy.c

- example time!

whodunit

- goal: change unreadable grid of red, white, and blue pixels into a readable grid
  - multiple ways to do this!

TODO

1. open file
2. read each scanline, pixel by pixel
3. change color of pixels in scanline
4. write scanline, pixel by pixel

copy.c

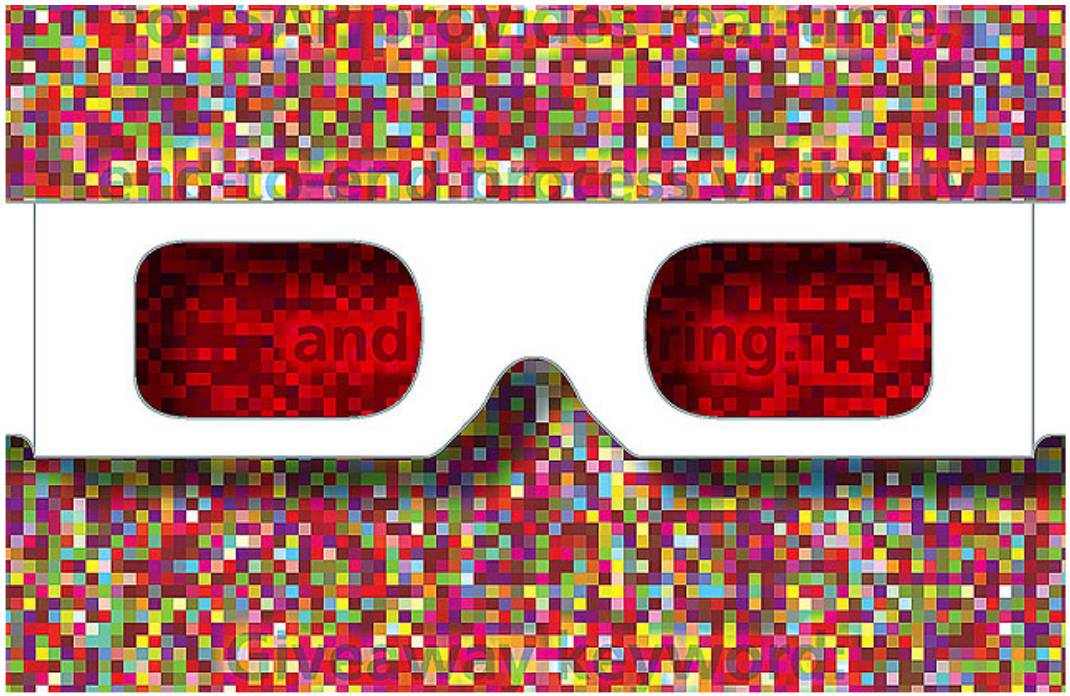
- start with `copy.c`
  - opening file, reading pixels, and writing pixels is already done!

TODO

1. ~~open file~~
2. ~~read each scanline, pixel by pixel~~
3. change color of pixels in scanline
4. ~~write scanline, pixel by pixel~~

Changing Colors

- current arrangement of colors is unreadable, so change some colors!
  - create a filter: only let red through by cranking down all blue and green in all pixels
  - good thing we can change red, blue, and green independently



TODO

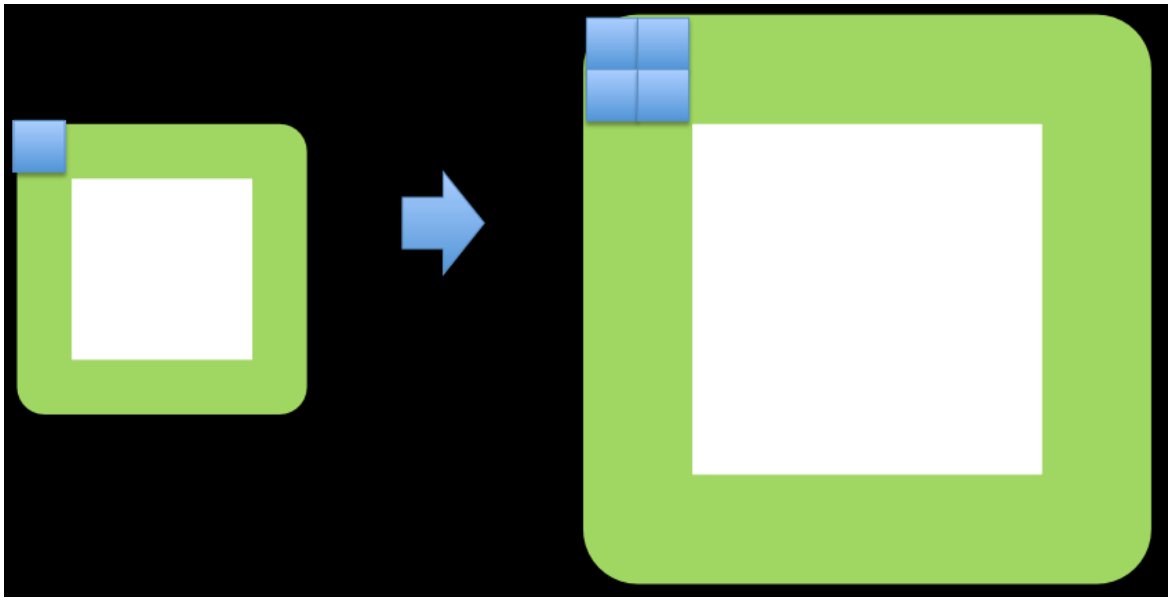
1. ~~open file~~
2. ~~read each scanline, pixel by pixel~~
3. ~~change color of pixels in scanline~~
4. ~~write scanline, pixel by pixel~~

resize

- goal: rather than copy image, scale image up by a factor of *n*
- `./resize 5 smiley.bmp`
  - each pixel needs to be repeated to create 5 pixels
  - each row needs to be repeated to create 5 rows

Resizing

- `./resize 2 small.bmp large.bmp`



TODO

1. ~~open file~~
2. update header info
3. ~~read each scanline, pixel by pixel~~
4. write each pixel *n* times
5. write each line *n* times

Headers

- new image means new header info!
- file size, image size, width, and height must change
- need to change both `structs`!

TODO

1. ~~open file~~
2. ~~update header info~~
3. ~~read each scanline, pixel by pixel~~
4. write each pixel *n* times
5. write each line *n* times

Resizing Horizontally

- `copy.c` reads in a single pixel, then writes a single pixel
- instead, we want to read a single pixel, then write that pixel multiple times
  - good thing we stored that pixel in a variable!
  - loops anyone?

Padding

- old image and new image might have different padding
  - need to recalculate, good thing we have a formula
- when reading, need to use original padding
  - remember, padding isn't an `RGBTRIPLE`, so we can't try to `fread` padding
- when writing, need to use newly calculated padding
  - else we write the wrong amount of padding, and our image fails :\

TODO

1. ~~open file~~
2. ~~update header info~~
3. ~~read each scanline, pixel by pixel~~
4. ~~write each pixel *n* times~~
5. write each line *n* times

Resizing Vertically

- now, we need to write each line $n - 1$ more times
  - but, `copy` forgets each pixel as soon as it writes it
- remember pixels in an array
  - each element in array can be a single `RGBTRIPLE`
  - write array block by block, or all at once
- use `fseek`
  - write a line, `fseek` back to the beginning of the line, and repeat

Variable-Sized Arrays


```
int n = 5;
int array1[n];
int array2 = malloc(n * sizeof(int));
// FREEING ARRAYS IS SERIOUS BUSINESS
free(array2);
```

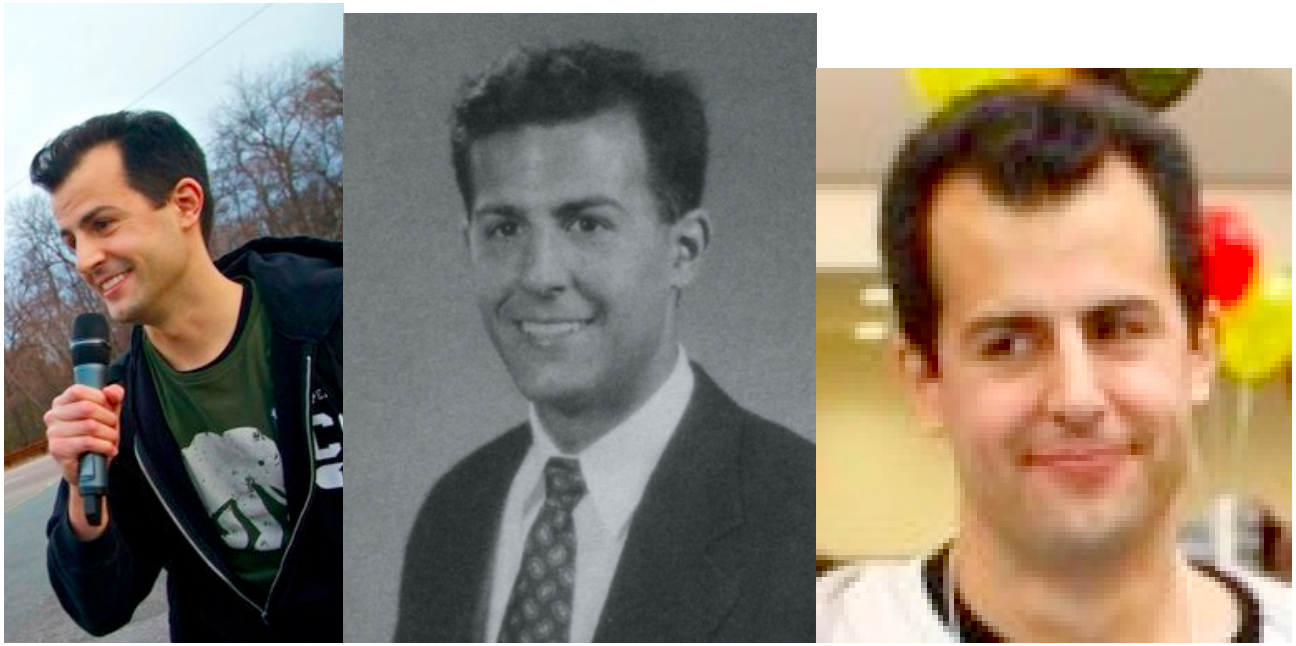
TODO

1. ~~open file~~
2. ~~update header info~~
3. ~~read each scanline, pixel by pixel~~
4. ~~write each pixel *n* times~~
5. ~~write each line *n* times~~

recover

- goal: recover 37 images from a corrupt CF card
  - someone needs to teach David computers

card.raw



TODO

1. open `card.raw`
2. determine start of new image
3. determine filename
4. write all bytes of image to the same file

card.raw

- located in `/home/cs50/pset5/card.raw`
- hard-code this value, no need for command-line parsing
  - `#define`, ahem

TODO

1. ~~open `card.raw`~~
2. determine start of new image
3. determine filename
4. write all bytes of image to the same file

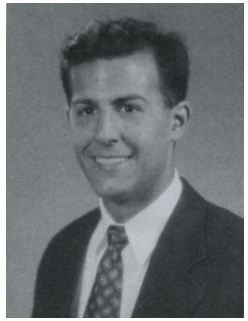
JPEGs

- JPEGs are still just a sequence of bytes
- start with either:
  - `0xff 0xd8 0xff 0xe0`
  - `0xff 0xd8 0xff 0xe1`
- stored contiguously on the CF card

card.raw


```
0xff 0xd8
0xff 0xe0
```

```
0xff 0xd8
0xff 0xe1
```



```
0xff 0xd8
0xff 0xe0
```

TODO

1. ~~open `card.raw`~~
2. ~~determine start of new image~~
3. determine filename
4. write all bytes of image to the same file

sprintf

- `printf` writes to standard output, `sprintf` writes to a `char *`
- `sprintf(array, "coolness: %d", 10);`
- filenames must be in the form `###.jpg`
  - good thing that's always the same number of characters!
- JPEGs named in the order you find them, starting at `000`

TODO

1. ~~open `card.raw`~~
2. ~~determine start of new image~~
3. ~~determine filename~~
4. write all bytes of image to the same file

Blocks

- JPEGs organized into 512-byte blocks
  - read file 512 bytes at a time instead of 3 bytes (aka 1 BMP pixel)

Reading Blocks

- we can also `fread` into arrays
  - arrays and structs are both contiguous in memory

```
BYTE array[5];
fread(array, sizeof(BYTE), 5, inptr);
```

Writing Blocks

- once we find an image, we can `fwrite` into the same file until we find the start of another image
  - then, we need to start `fwrite`-ing into the next file
  - don't forget to `fwrite` the block containing the start sequence

Finishing Up

- `feof(inptr)`
  - returns a boolean: have we reached the end of a file?
  - don't know how many bytes the card is, so loop until entire file is read

TODO

1. ~~open `card.raw`~~
2. ~~determine start of new image~~
3. ~~determine filename~~
4. ~~write all bytes of image to the same file~~

One More Thing

“Yesterday my sister accidentally formatted her camera’s SD card and lost a year’s worth of memorable photos. (She unfortunately isn’t the best at backing up her data.) This situation reminded me of Pset-5 so I thought I would try to run her SD card through the "recover.c" program I wrote all the way back in October. So after 4 hours of figuring out how to create a .raw image from the formatted the SD card and installing/configuring the CS 50 Virtual Box, **I managed to run the forensic image through my program and recover all 1027 of my sister’s photos.**”