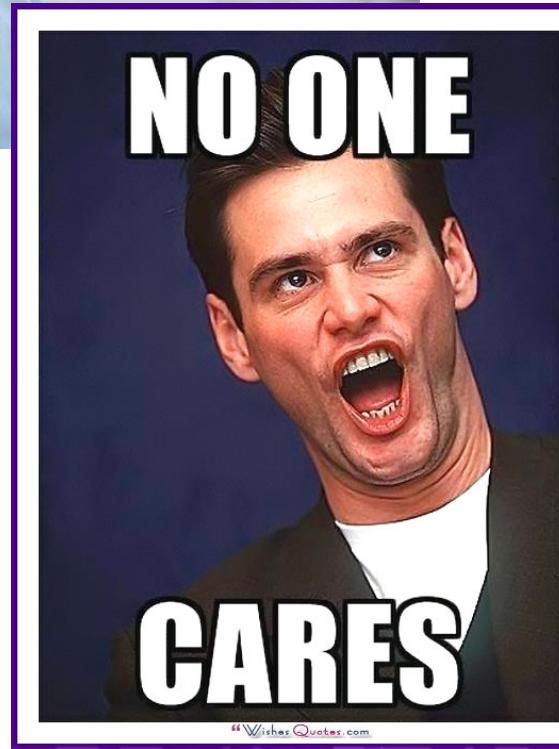


# COMP 206 – Introduction to Software Systems

Lecture 13 – Working with BMP Images

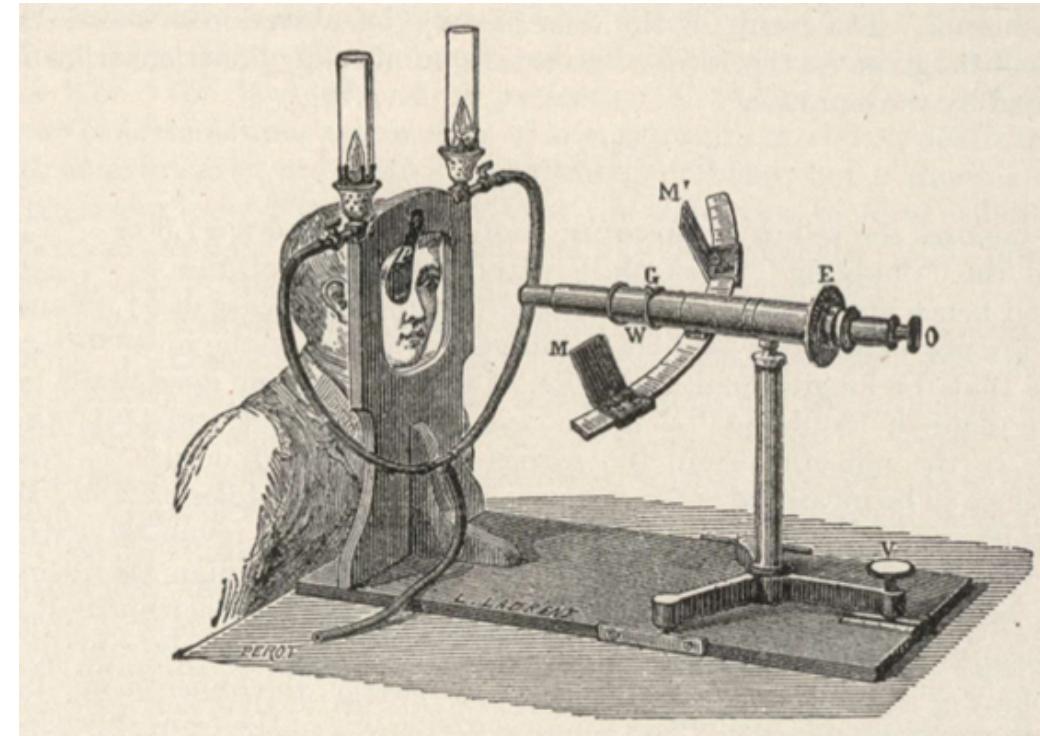
October 19, 2018

# We are ready for (really useful) images!



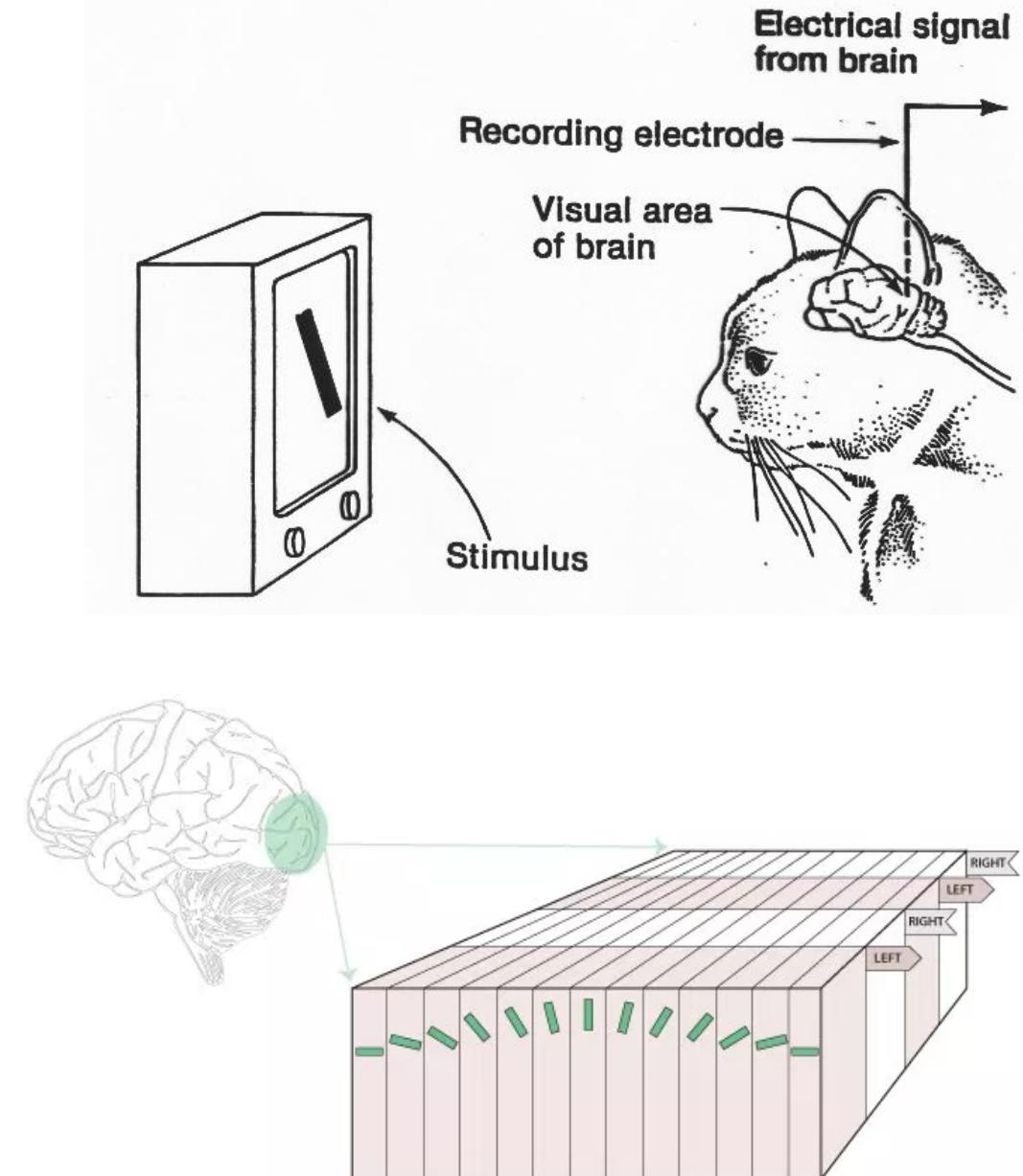
# Human Vision (not testable)

- How do people see?
  - One of the fundamental questions in early science. Major player: [Hermann von Helmholtz](#) 1821 – 1894. He understood that the eye has a lens which projects light using physical laws. What happens afterwards?



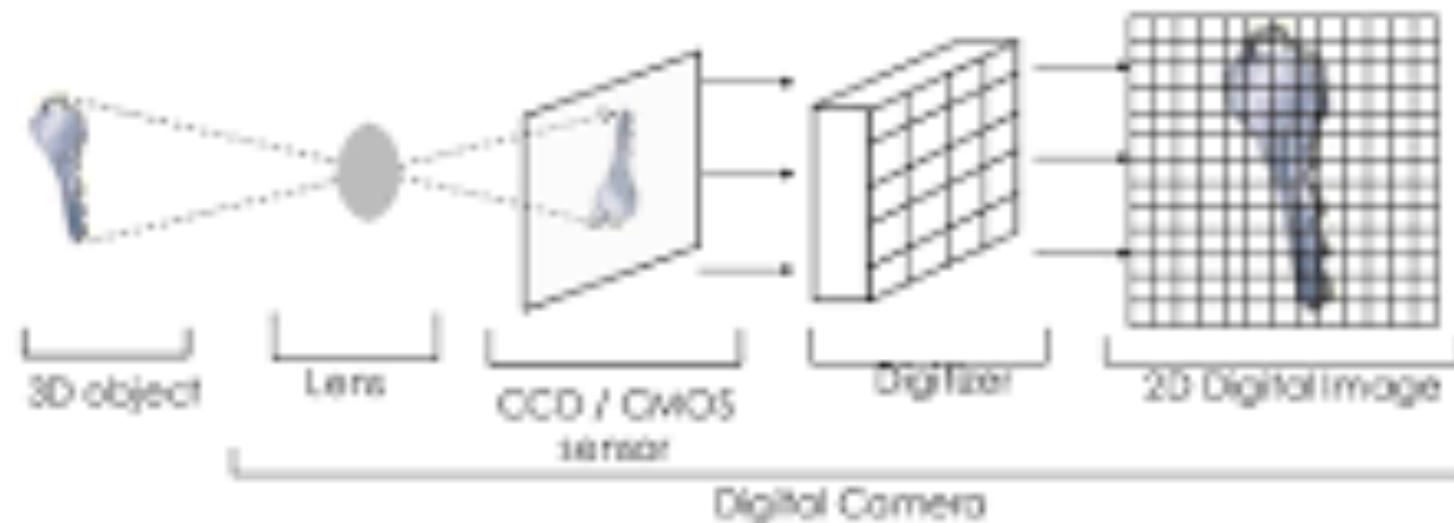
# Human Vision (not testable)

- How do people see?
  - Now we know about the retina with light-sensitive rods and cones in precise pattern.
  - Optical nerve transmits sensed light to visual cortex arranged in "retino-topic" fashion (like a grid) for several layers.  
Nobel prize for [David H. Hubel](#) and [Torsten N. Wiesel](#) in 1981.



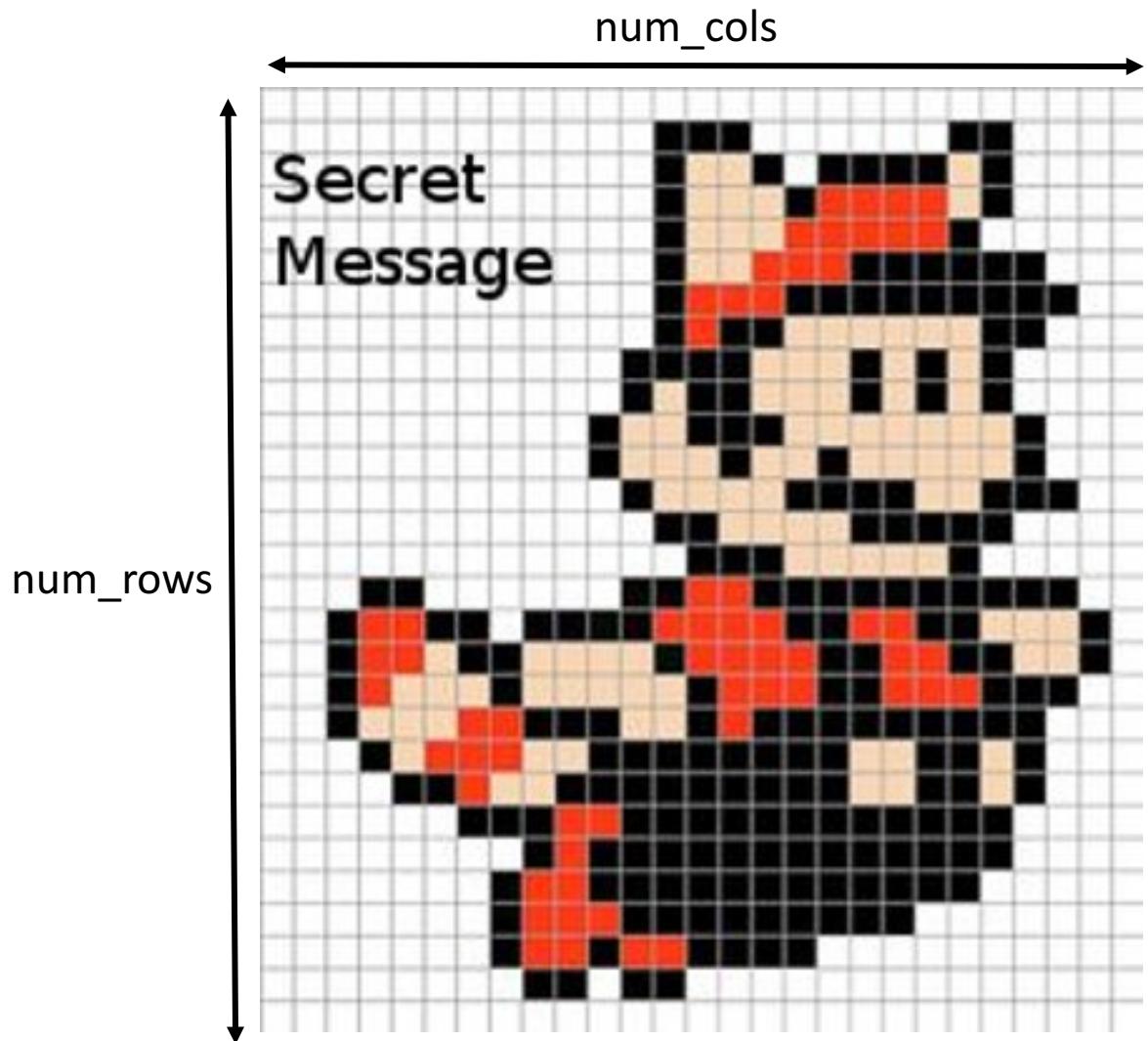
# How do cameras form images?

- Project light with a lens that has similar behavior to our eye.
- Instead of a retina, light sensitive electronics (CCD or CMOS), count arriving photons. Each is tuned for a color, we call Red, Green, or Blue. The RGB values at one spot are called a *pixel*.
- Each pixel is read off as 3 integer values (binary memory!)



# How to store images as a file on disk?

- An image is a 2D grid of pixels:
  - $\text{num\_rows}$  = height
  - $\text{num\_cols}$  = width
  - $\text{num\_colors}$ : how many per pixel could be 3 for RGB, 1 for b/w, or 4 for RGBA (alpha = transparency)
  - Bits per pixel: what size of integer is needed to store each?
- 2 additional types of data:
  - A header, holds information fields such as the image size, compression, color depth
  - Padding, almost always present to align the elements into 4 or 8 byte boundaries (details coming)

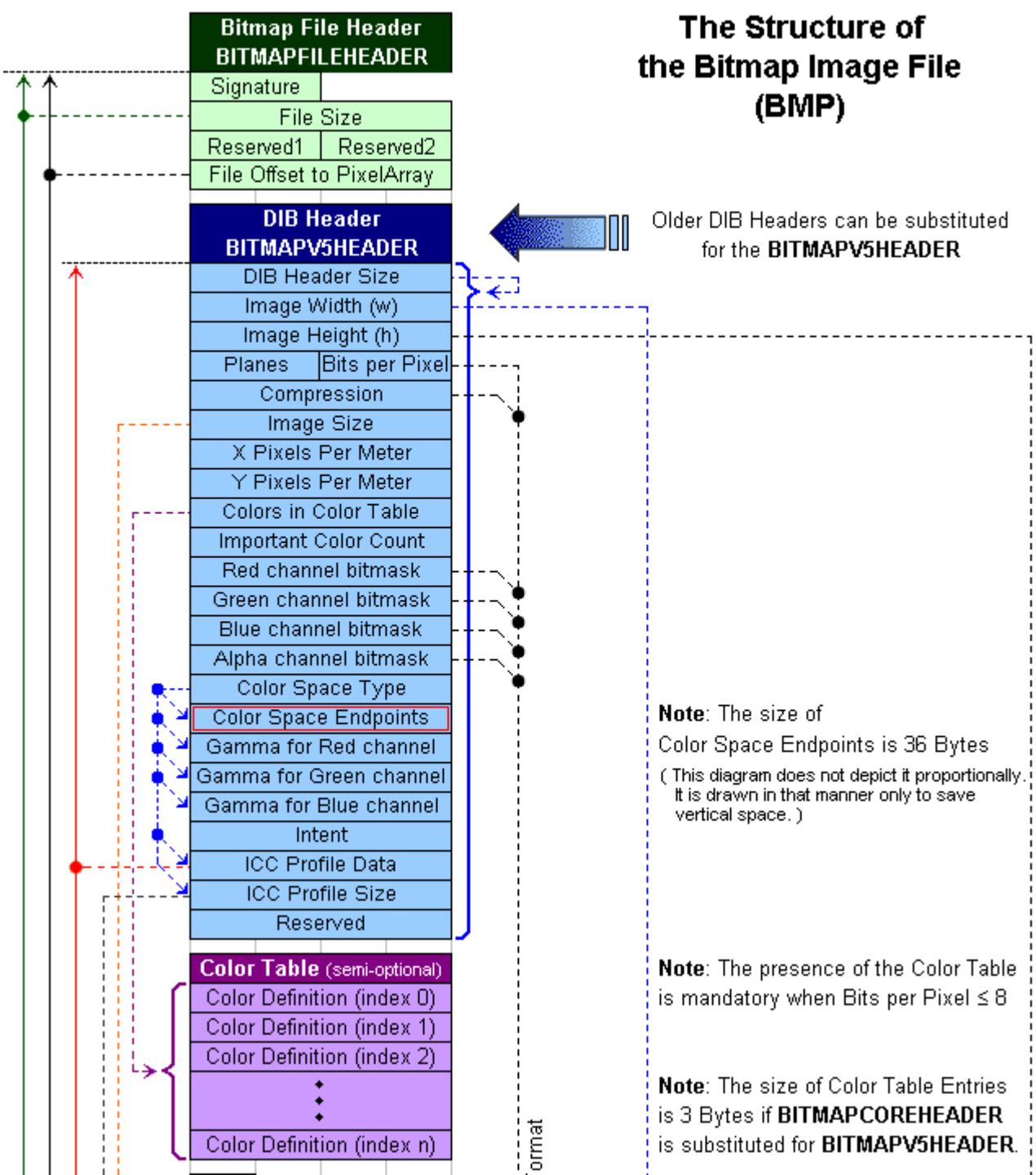


# Many image file formats

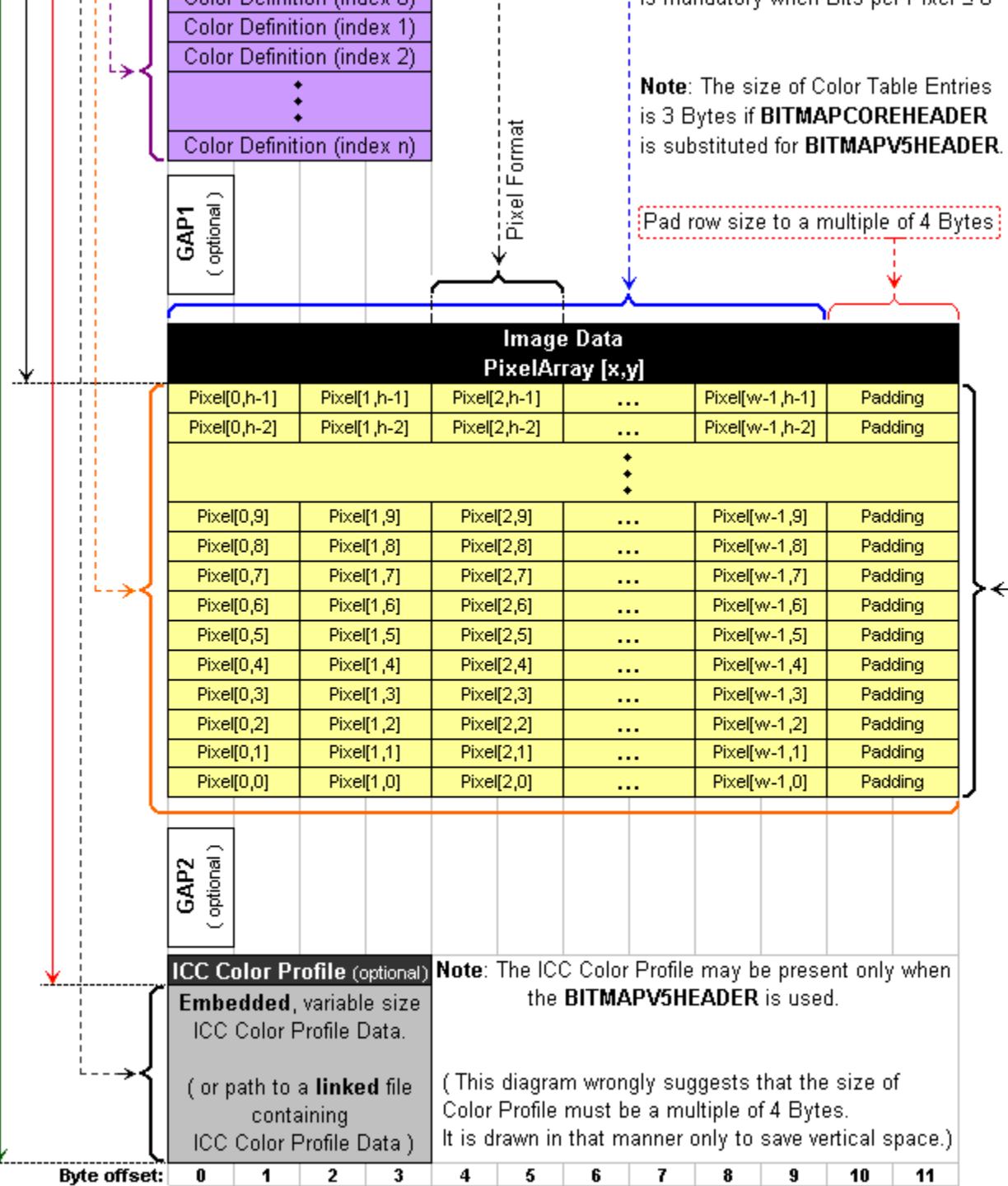
- BMP: "Windows bitmap"
- JPG: Joint Photographic Experts Group
- PNG: Portable Network Graphics
- TIFF: Tagged Image File Format
- GIF: Graphics Interchange Format
- Of these, only BMP is testable in 206 and will be used in A3.



# Bitmap File (BMP) Example first half



# Bitmap File (BMP) Example second half



# How can we read a BMP file using C?

- What works well:
  - Check the magic number:
    - If it matches very likely it follows the rules
  - File size field: makes it easy to access all of the data
  - Width and height, allows finding a specific pixel
  - Opening with code like “rb”
- What we must avoid:
  - Checking for AASCI code values: space, newline, etc
  - Attempting to use “atoi” “atof”, these are “aasci to ...”
  - If we open with “r” alone (no b), C will do some of this automatically and cause us problems.
  - fgets, fscanf also typically bad choices, mean to work with text

# Example

- Github:  
ExampleCode/  
Lecture13 folder
- Note "18" is the byte  
for width using the  
chart above
- Ensure you  
understand how to  
read the chart  
(needed for A3)

```
int main(){

    // Open a binary bmp file
    FILE *bmpfile = fopen( "utah.bmp", "rb" );

    if( bmpfile == NULL ){
        printf( "I was unable to open the file utah.bmp.\n" );
        return -1;
    }

    // Read the B and M characters into chars
    char b, m;
    fread( &b, 1, 1, bmpfile );
    fread( &m, 1, 1, bmpfile );

    // Print the B and M to terminal
    printf( "The first byte was: %c.\n", b );
    printf( "The second byte was: %c.\n", m );

    // Read the overall file size
    unsigned int overallFileSize;
    fread( &overallFileSize, 1, sizeof(unsigned int), bmpfile );
    printf( "The size was: %d.\n", overallFileSize );

    // Rewind file pointer to the beginning and read the entire contents.
    rewind(bmpfile);

    char imageData[overallFileSize];
    if( fread( imageData, 1, overallFileSize, bmpfile ) != overallFileSize ){
        printf( "I was unable to read the requested %d bytes.\n", overallFileSize );
        return -1;
    }

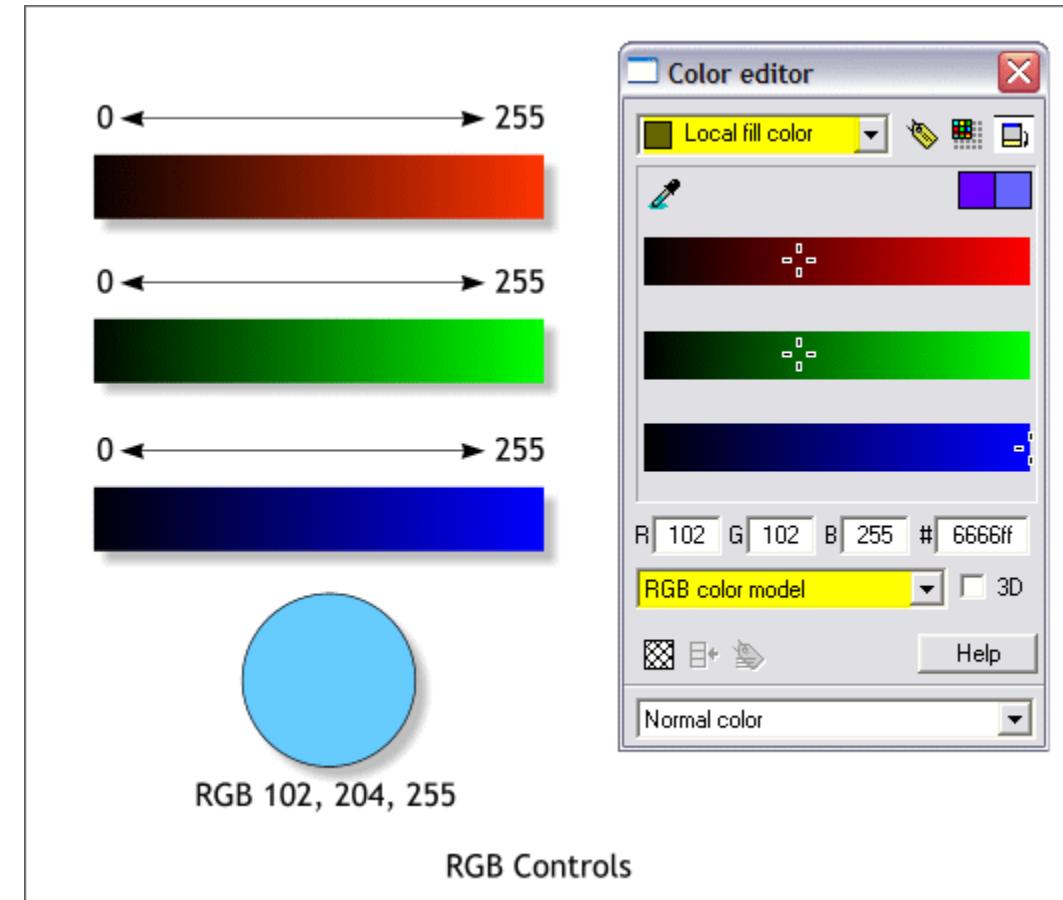
    // Read the width size into unsigned int (hope = 500 since this is the width of utah.bmp)
    unsigned int* wp = (unsigned int*)(imageData+18);
    unsigned int width = *wp;

    // Print the width size to terminal
    printf( "The width is: %d.\n", width );

    return 0;
}
```

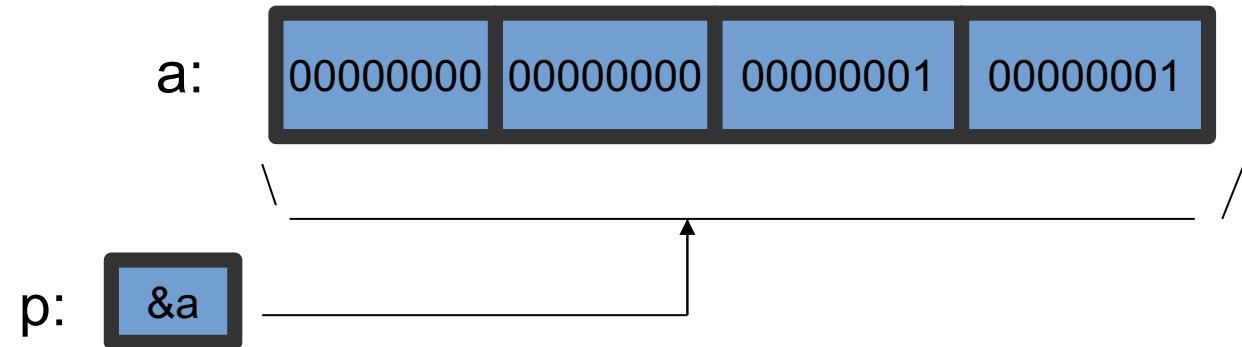
# Now that we have the data...

- Each color of each pixel is stored as an integer between 0 and 255 (one byte):
  - Easiest way to work with these in C: represent each as an unsigned char
- Other items such as the length and width are 4 byte integers
- It's time to get brave and learn some new tools to work with mixed memory!



# Normal pointer use

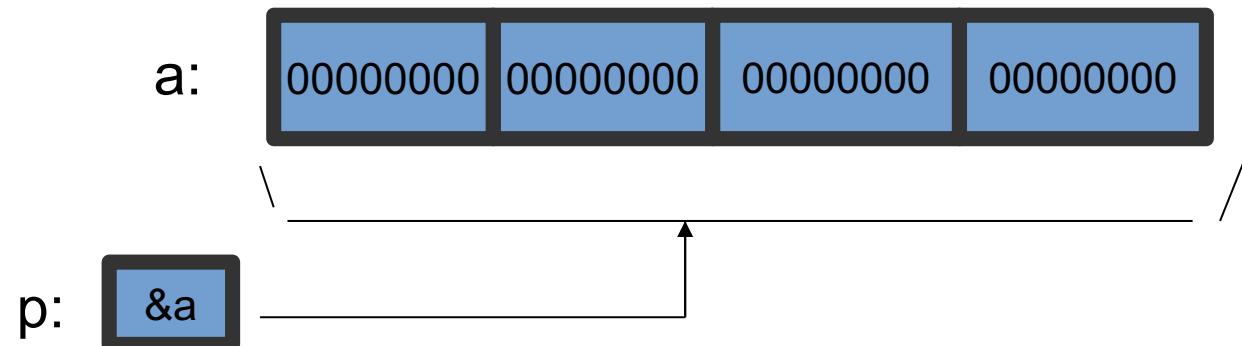
- `int *p;`      -> a pointer to an integer value
- `int a = 257;` -> an integer variable
- `p = &a;`      -> p now holds the **address of a**  
(points to a)



All is well here. But what if we change the types?

# Pointers can re-interpret memory

- `int *p;` → a pointer to an integer value
- `char a[4];` → a character array (string)
- `memset( a, '\0', 4 );`
- `p = (int*)a;` → p now holds the **address of a** (points to a)



- This is OK, and treats the whole string as integer data (equivalent integer value 0)

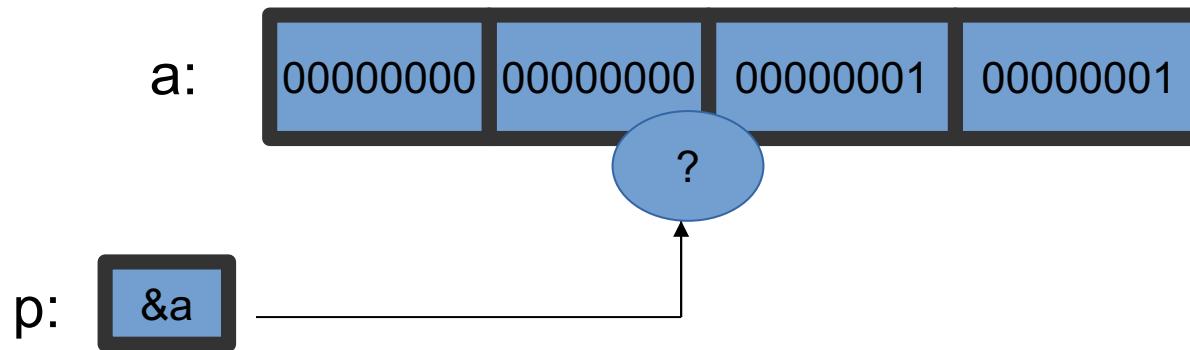
# Example Code

- ExampleCode/Lecture13-ImageFiles/pointer\_conversion.c
- Try this out, think about changing values in the C string – what happens to the pointer? This can all seem weird, so it's important you really try it yourself!

# What about the other direction?

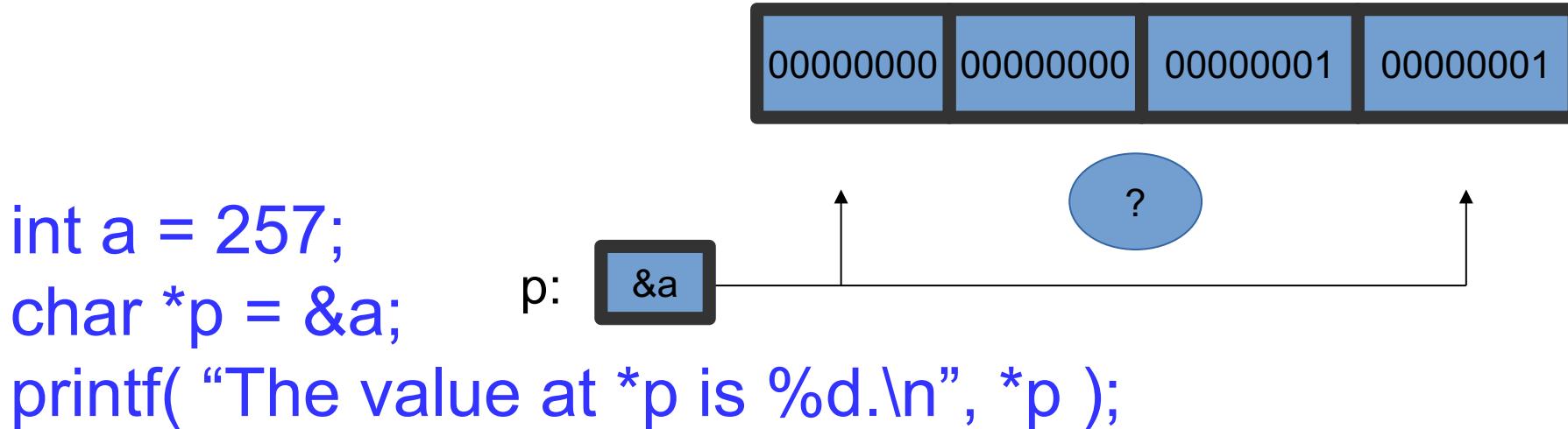
- `char *p;`      -> a pointer to a character value
- `int a = 257;`    -> an integer variable
- `p = (char*)&a;` -> p now holds the **address of a**,

but will interpret it as character memory, 1 byte  
only, when dereferenced!



- Think carefully about what character byte p will address.  
What will the value of `*p` be?

# Pointers and Type Conversion

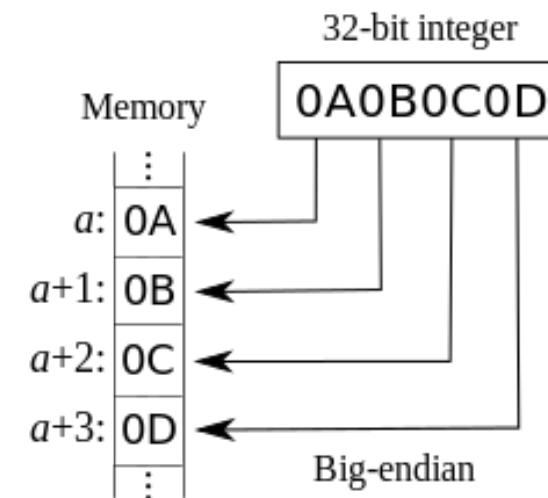
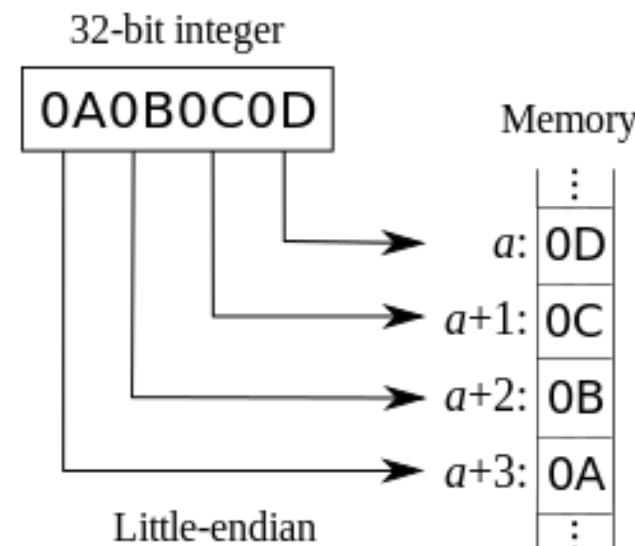


The output of this program depends on “endianness”. Sample code GitHub:

[endianness\\_test.c](#)

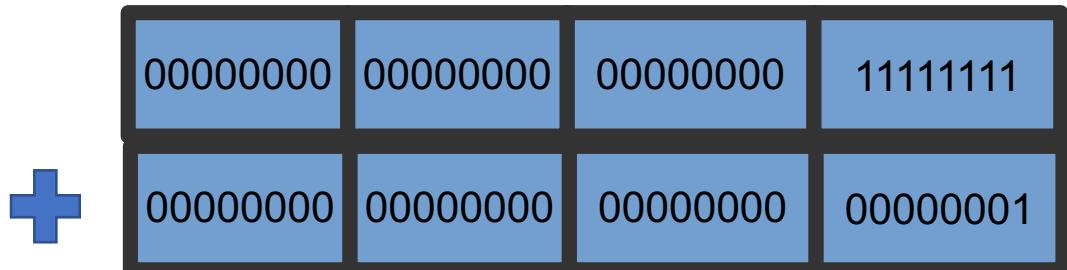
# Word Endianess

- The order that bytes within an integer are stored in memory is a convention, named Endianess, and there is no right answer.
- Most systems we deal with will be Little-endian, but there are major exceptions (the Sun company)
- It is always better to check than to assume

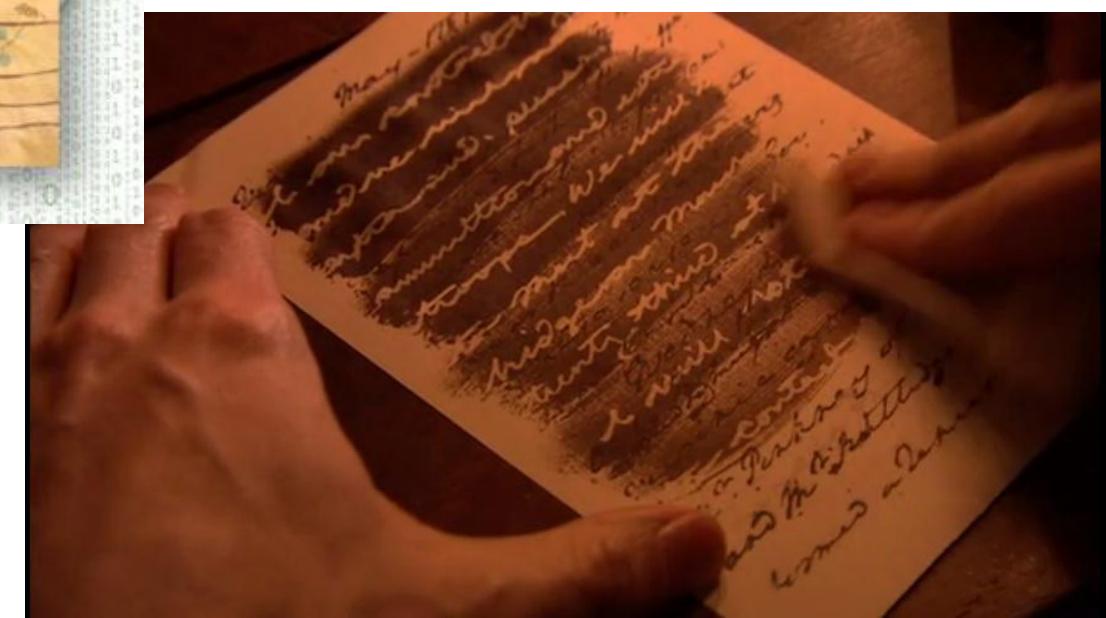


# Endianess intuition helper

- Little or big? Name determined by the "significance" of the byte at the first address:
  - Little: the least significant comes first
  - Big: the most significant comes first
- Humans always write numbers in Big Endian, why would most computers use Little?
  - Think about doing addition with carry-over
  - We process right to left and "carry"
  - Computer add a lot, and are most efficient accessing memory "in order"



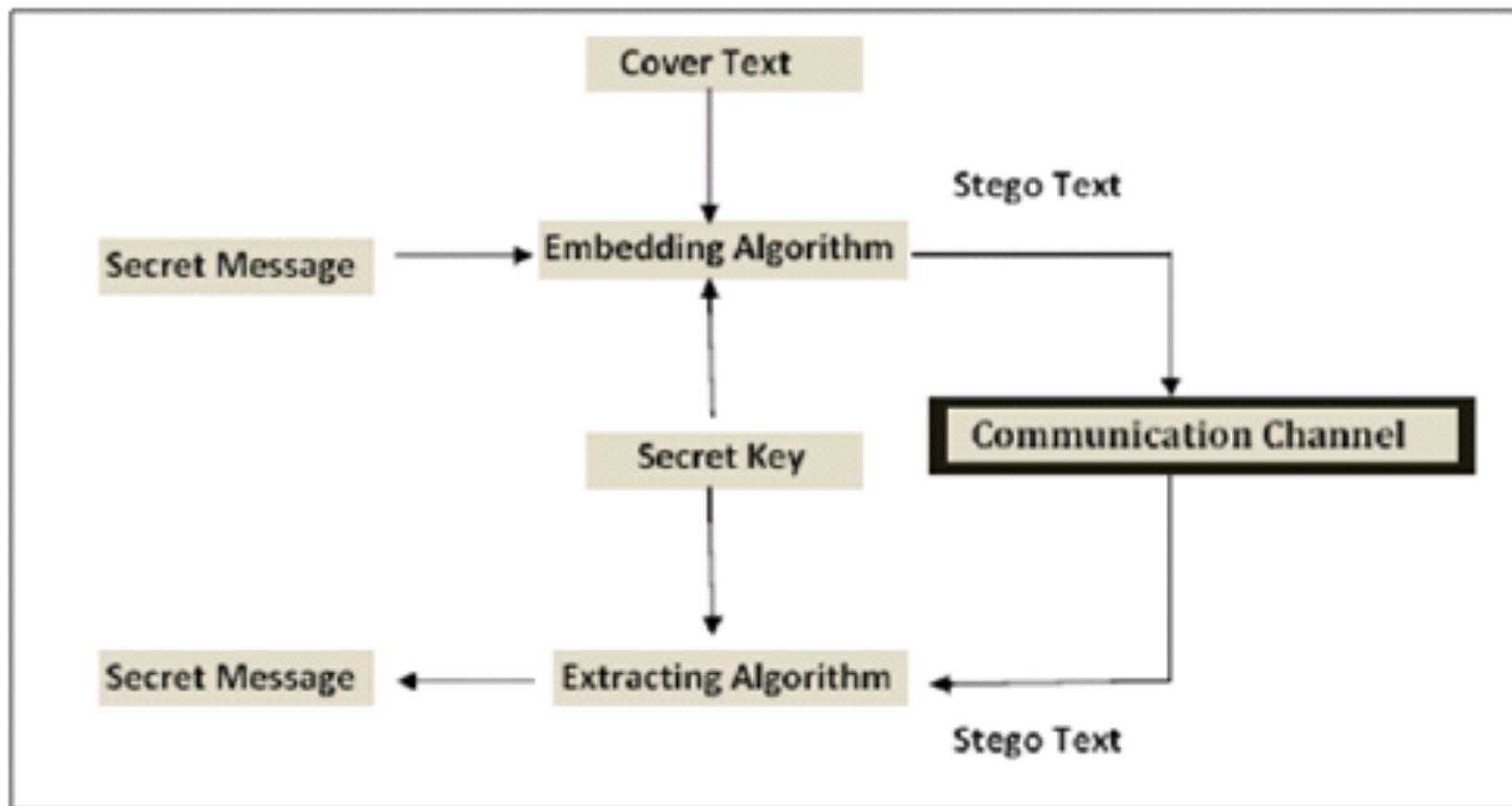
# A motivating example: steganography



# Steganography

- ...is concerned with concealing the fact that a secret message is being sent as well as concealing the contents of the message.
- Whereas cryptography is the practice of protecting the contents of a message alone, steganography is concerned with concealing the fact that a secret message is being sent as well as concealing the contents of the message.

# Diagram of a Steganography Software System



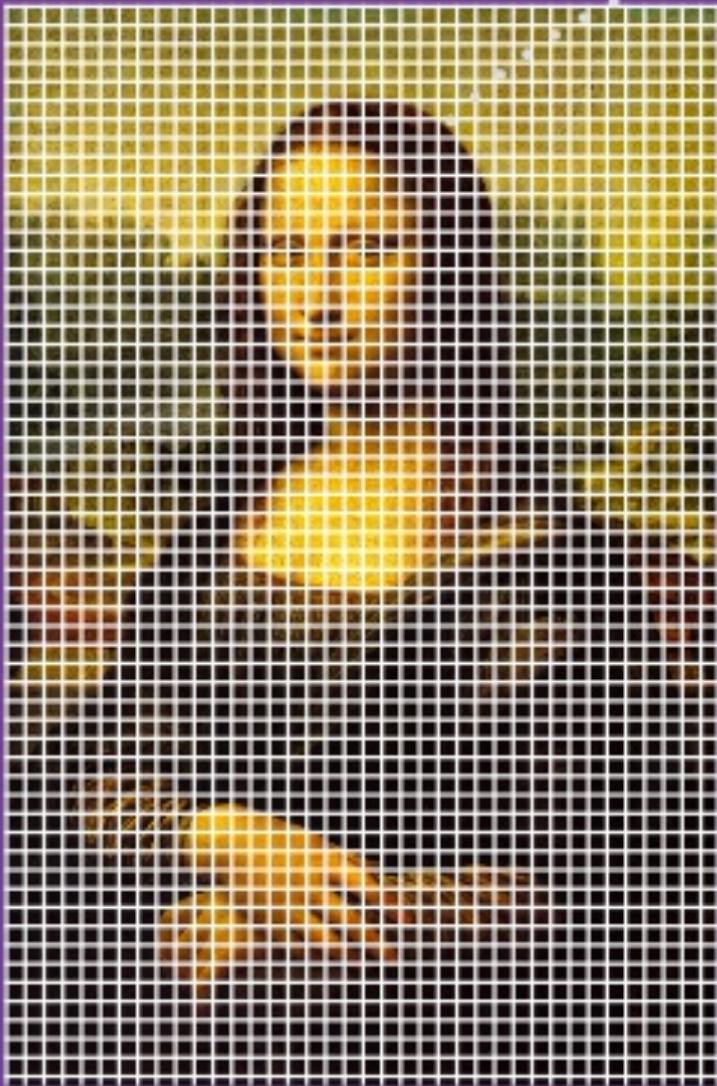
# Live Demo!

## steg\_encoder.c and steg\_decoder.c

- Two files posted in this lecture's folder on GitHub
- We have not yet covered everything needed to understand them, but we will demo quickly to show what we're after
- We'll have covered all the details by the end of this week

# Digital Steganography

## LSB IN IMAGES



144 141 81

10010000 10001101 01010001

**Hidden message: 101001...**

145 140 81

1001000**1** 1000110**0** 0101000**1**

146 142 81

100100**10** 100011**10** 010100**01**

# Elements Required for In-Image Steganography

- Read and write binary image data
- View the text string in binary form, so we can access one bit at a time
- Ability to modify only a single bit (the LSB) of each pixel
- Ability to extract the LSB again for decoding

# Bit-wise Operations

Shifts:

- `bit_arg<<shift_arg`
  - Shifts bits to of bit\_arg shift\_arg places to the left -- equivalent to multiplication by  $2^{shift\_arg}$
- `bit_arg>>shift_arg`
  - Shifts bits to of bit\_arg shift\_arg places to the right -- equivalent to integer division by  $2^{shift\_arg}$

Bit-wise logic:

- `left_arg & right_arg`
  - Takes the bitwise AND of left\_arg and right\_arg
- `left_arg | right_arg`
  - Takes the bitwise OR of left\_arg and right\_arg
- `left_arg ^ right_arg`
  - Takes the bitwise XOR of left\_arg and right\_arg (one or the other but not both)
- `~arg`
  - Takes the bitwise complement of arg

# Bitwise Logical Operators

- Each applies a *truth table* to the bits in its arguments, one at a time
- Logical AND and logical OR truth tables:

&	<table border="1" style="border-collapse: collapse; width: 100%;"><tbody><tr><td style="padding: 5px;">0</td><td style="padding: 5px;">0</td><td style="padding: 5px;">1</td><td style="padding: 5px;">1</td></tr><tr><td style="padding: 5px;">0</td><td style="padding: 5px;">1</td><td style="padding: 5px;">0</td><td style="padding: 5px;">1</td></tr></tbody></table>	0	0	1	1	0	1	0	1	<table border="1" style="border-collapse: collapse; width: 100%;"><tbody><tr><td style="padding: 5px;">0</td><td style="padding: 5px;">0</td><td style="padding: 5px;">1</td><td style="padding: 5px;">1</td></tr><tr><td style="padding: 5px;">0</td><td style="padding: 5px;">1</td><td style="padding: 5px;">0</td><td style="padding: 5px;">1</td></tr></tbody></table>	0	0	1	1	0	1	0	1	
0	0	1	1																
0	1	0	1																
0	0	1	1																
0	1	0	1																
	<hr style="border: 1px solid blue; margin-top: 5px; margin-bottom: 5px;"/>	<hr style="border: 1px solid blue; margin-top: 5px; margin-bottom: 5px;"/>																	
	<table border="1" style="border-collapse: collapse; width: 100%;"><tbody><tr><td style="padding: 5px;">0</td><td style="padding: 5px;">0</td><td style="padding: 5px;">0</td><td style="padding: 5px;">1</td></tr></tbody></table>	0	0	0	1	<table border="1" style="border-collapse: collapse; width: 100%;"><tbody><tr><td style="padding: 5px;">0</td><td style="padding: 5px;">1</td><td style="padding: 5px;">1</td><td style="padding: 5px;">1</td></tr></tbody></table>	0	1	1	1									
0	0	0	1																
0	1	1	1																

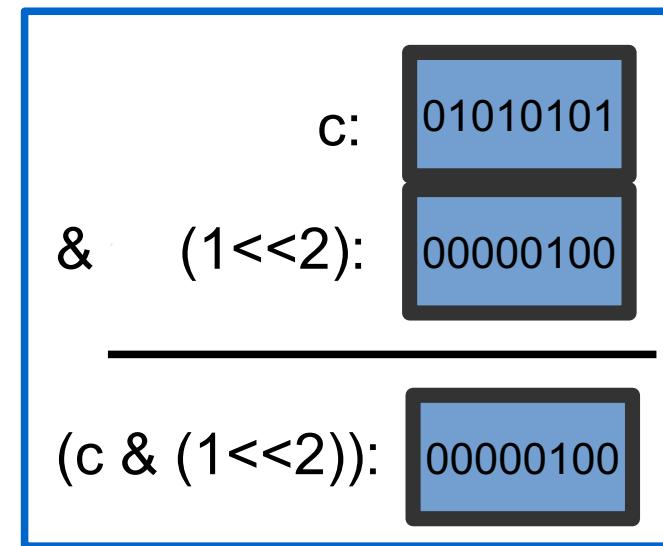
- When you apply a & b, these operations are applied to all of the bits in a and b, 1 bit at a time

# Bitwise Example #1

- Check the value of bit 3:

```
char c = 85;  
if( (c & (1<<2)) > 0 )  
    printf( "bit 3 of c is a 1!\n" );  
else  
    printf( "bit 3 of c is a 0!\n" );
```

- Larger example posted to Github:
  - [bit\\_reporting.c](#)



# More Bitwise Examples

- Set the value of bit 8 to 1
- Count the number of 1's
- Find the first 0 starting from the right

# Back to Steganography

- Recall: if we encode a string within the “lowest-order” bits of an image, it is nearly invisible
- We will examine the example code, posted on Github for this lecture
  - [steg\\_encoder.c](#)
  - [steg\\_decoder.c](#)



# Steganography

## Learning Outcomes

- Understand how our code interacts with external systems:
  - BMP file format: defines how our data is structured, forces our code to ignore the header, binary data format
  - Image viewer: displays our data, determines which parts of the data matter
  - Our program is called on the command-line, can interact with BASH functions.
    - **CHALLENGE #1:** How would you “hide” the contents of another file inside the image?

**MOSTLY COMPLETE!**

# Steganography

## Learning Outcomes

- Understand how the two pieces of our code interact with each other:
  - **CHALLENGE #2:** What if we wanted to change the bit used to hide our data?
  - What if the decoder wanted to switch to JPG data?
  - How does the decoder know the length of the message?
  - **CHALLENGE #3:** Could we make each side more robust to ensure the image and message are “proper”?