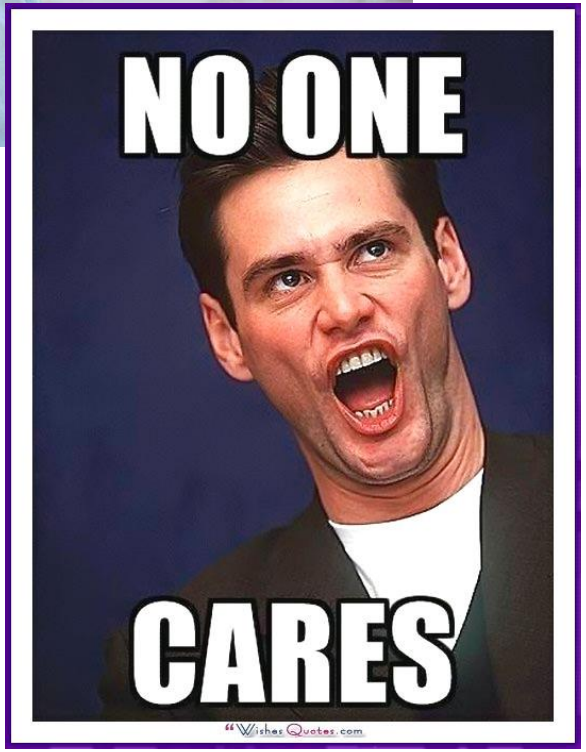
COMP 206 – Introduction to Software Systems

Lecture 13 – Working with BMP Images

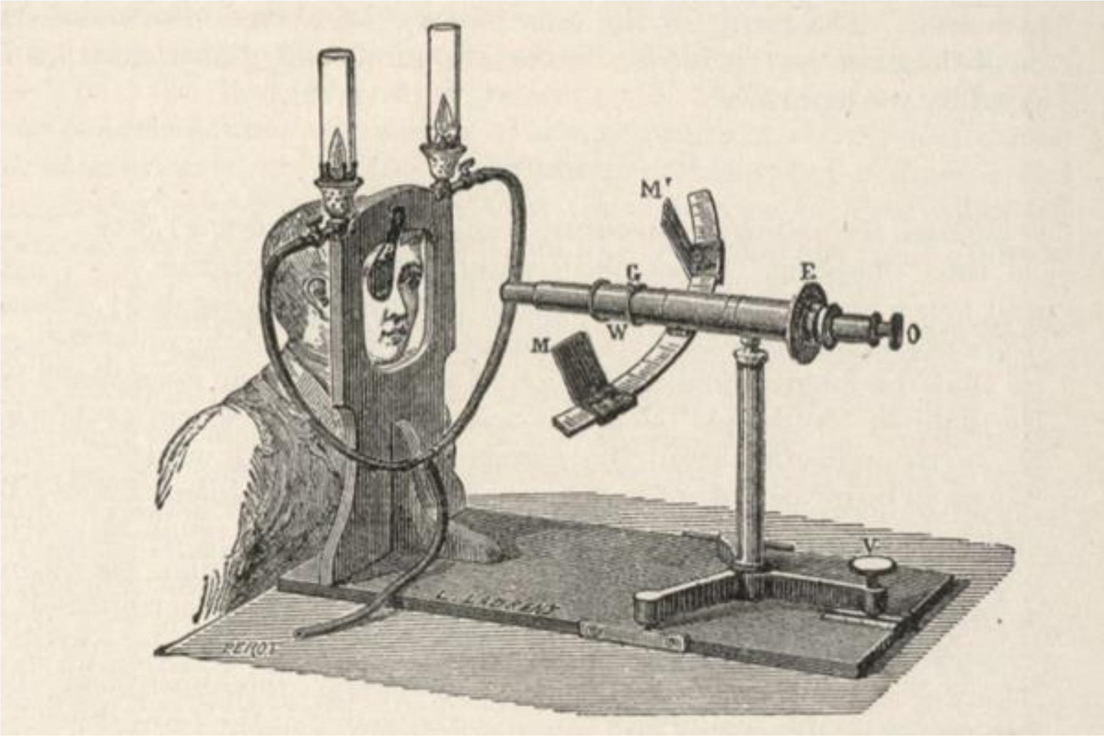
October 19, 2018

We are ready for (really useful) images!



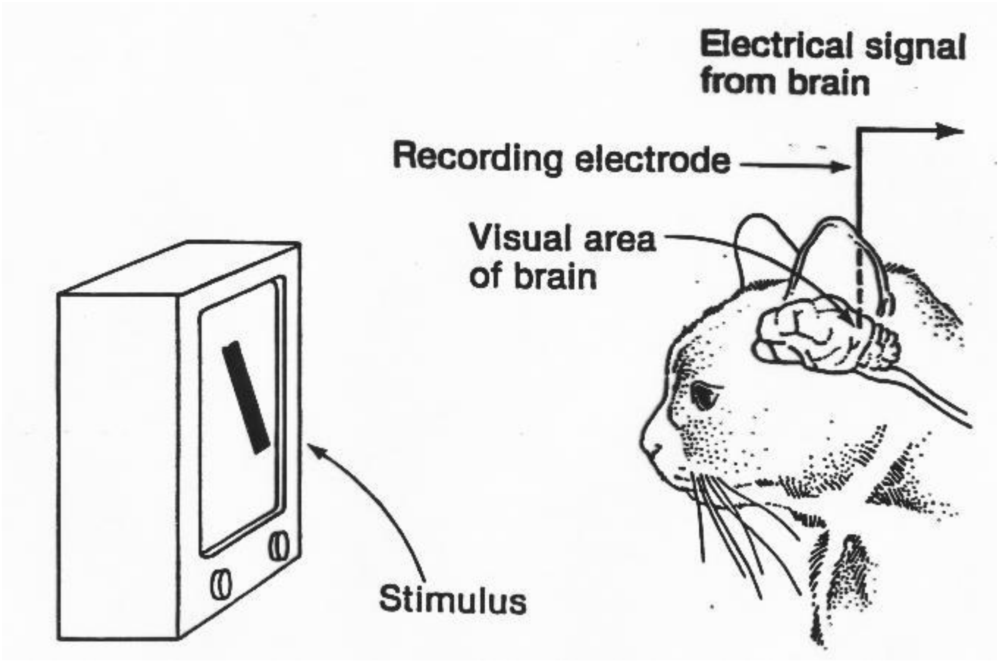
Human Vision (not testable)

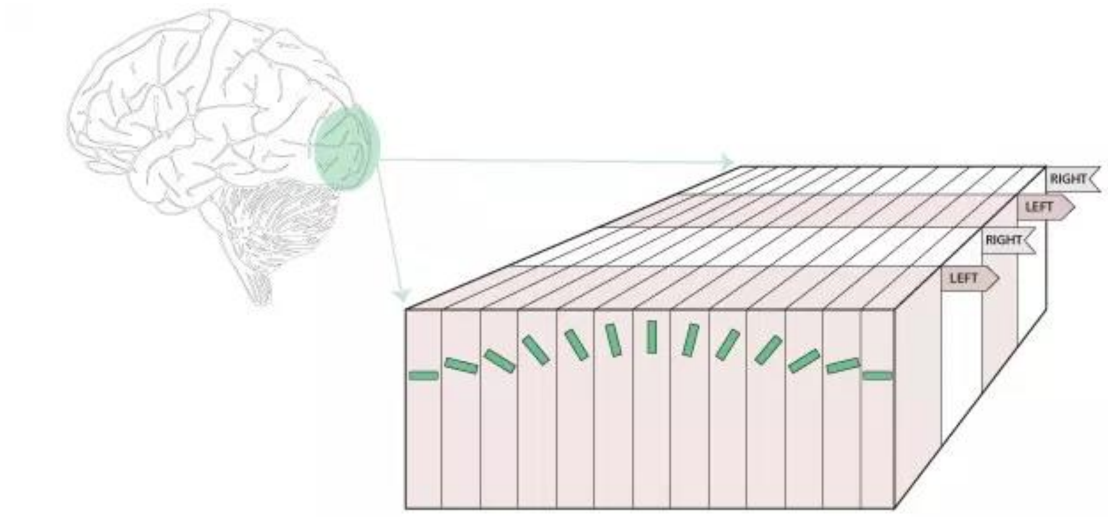
- How do people see?
  - One of the fundamental questions in early science. Major player: Hermann von Helmholtz 1821 – 1894. He understood that the eye has a lens which projects light using physical laws. What happens afterwards?



Human Vision (not testable)

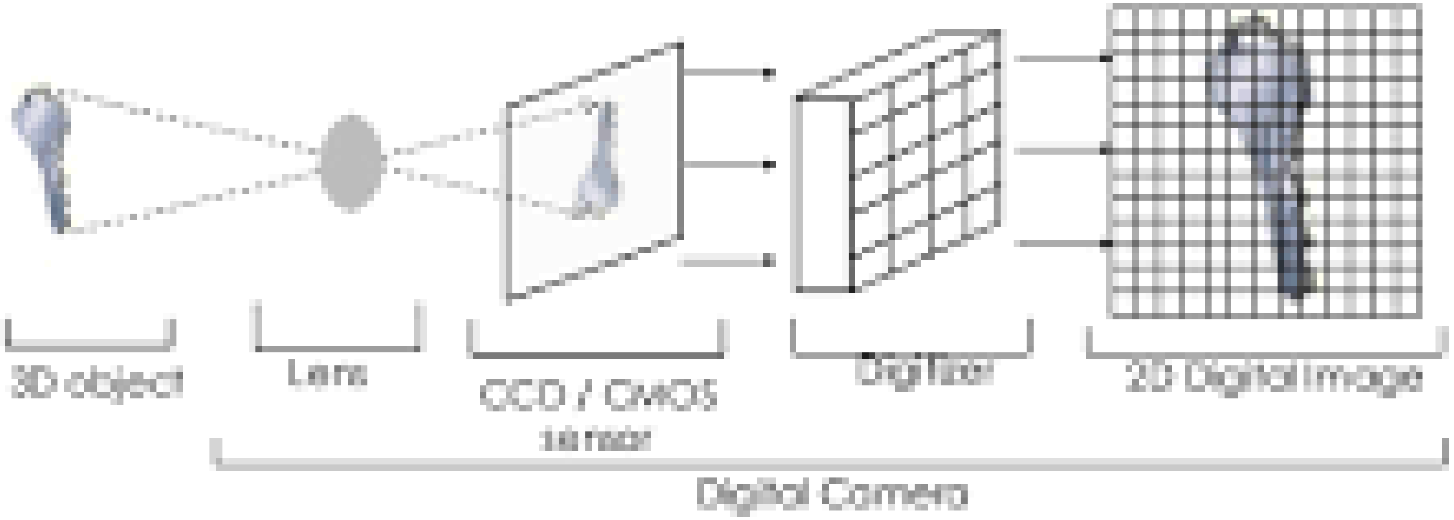
- How do people see?
  - Now we know about the retina with light-sensitive rods and cones in precise pattern.
  - Optical nerve transmits sensed light to visual cortex arranged in "retino-topic" fashion (like a grid) for several layers. Nobel prize for David H. Hubel and Torsten N. Wiesel in 1981.





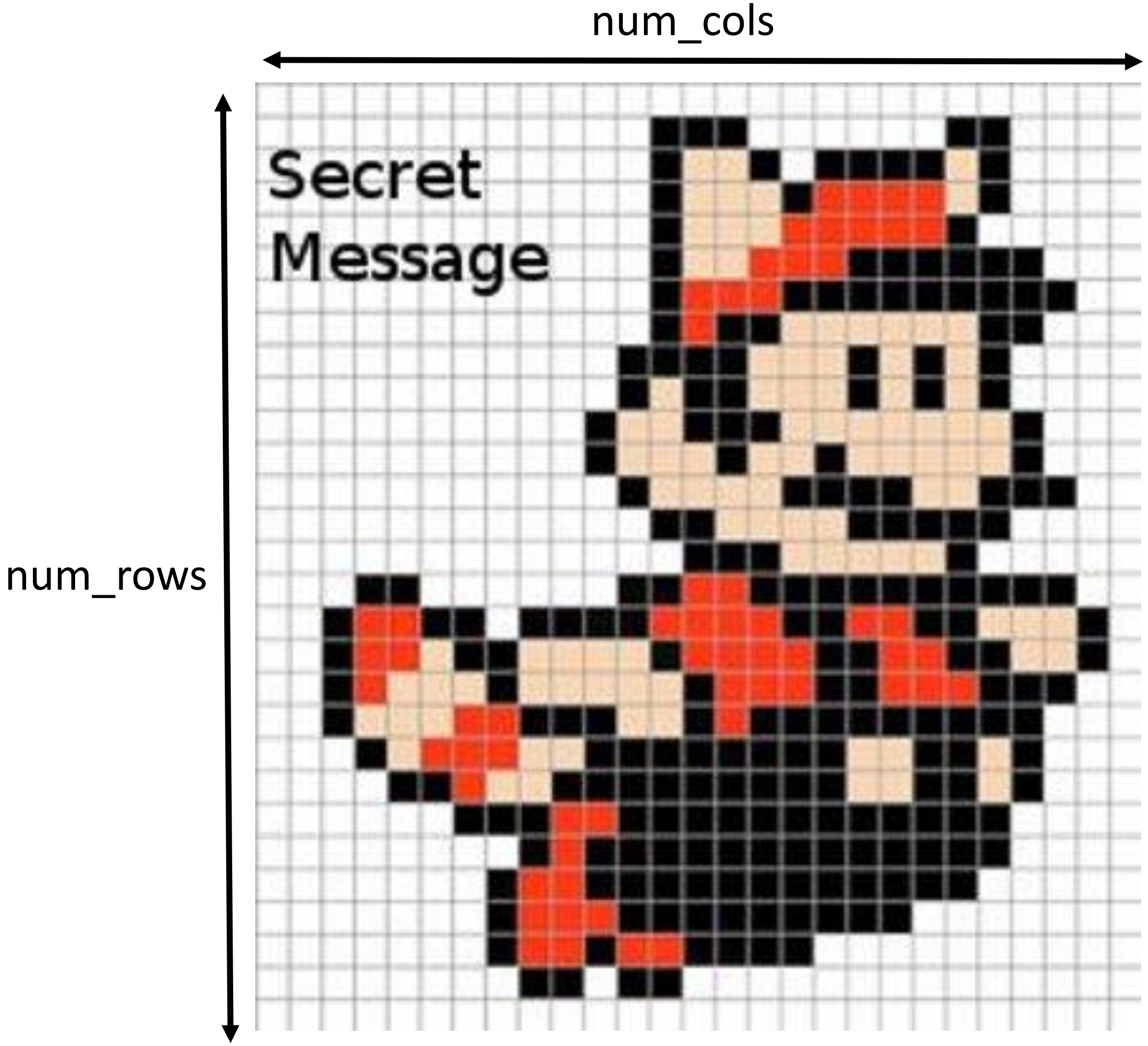
How do cameras form images?

- 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:
  - num_rows = height
  - num_cols = width
  - 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)

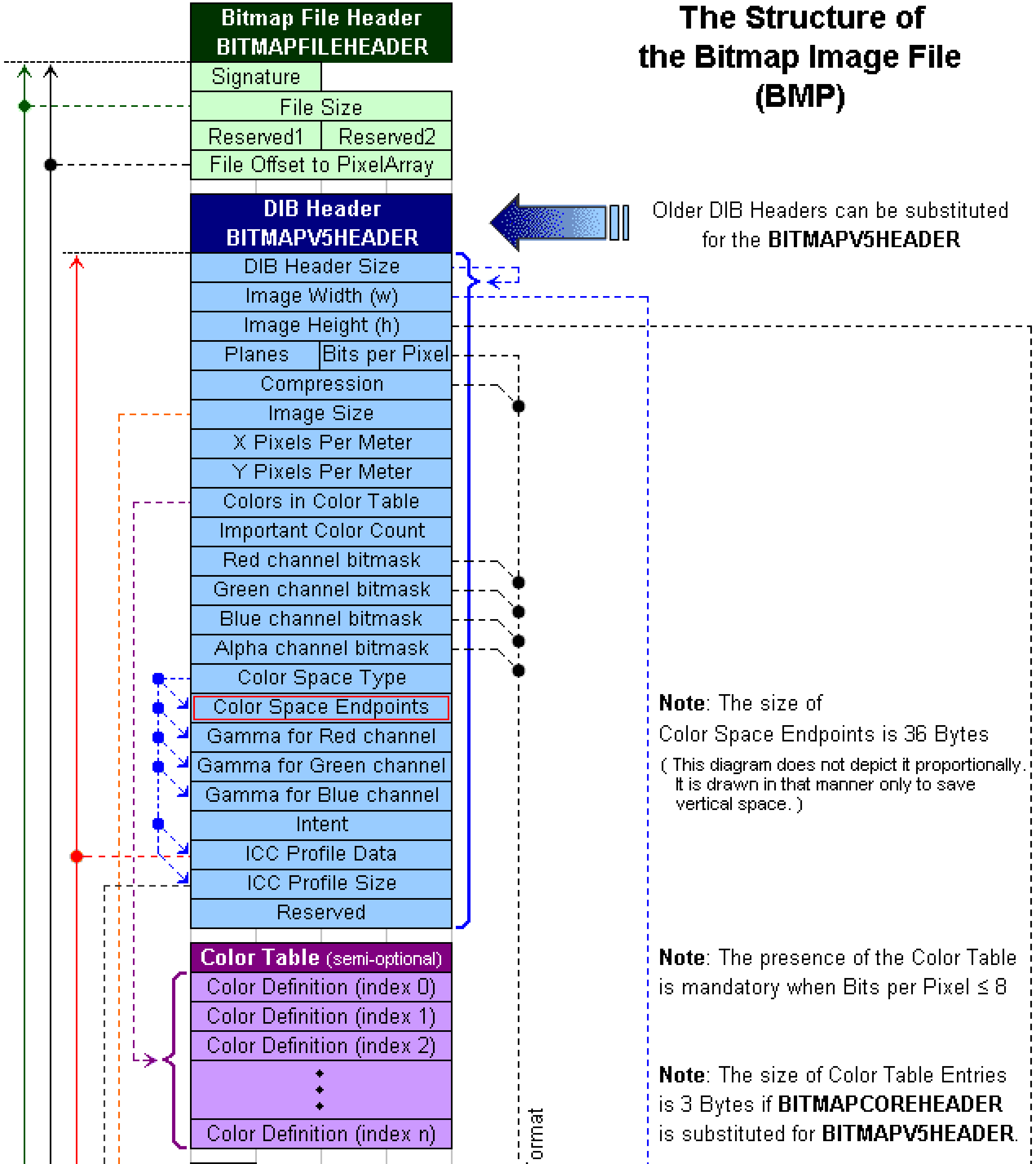


Many image file formats

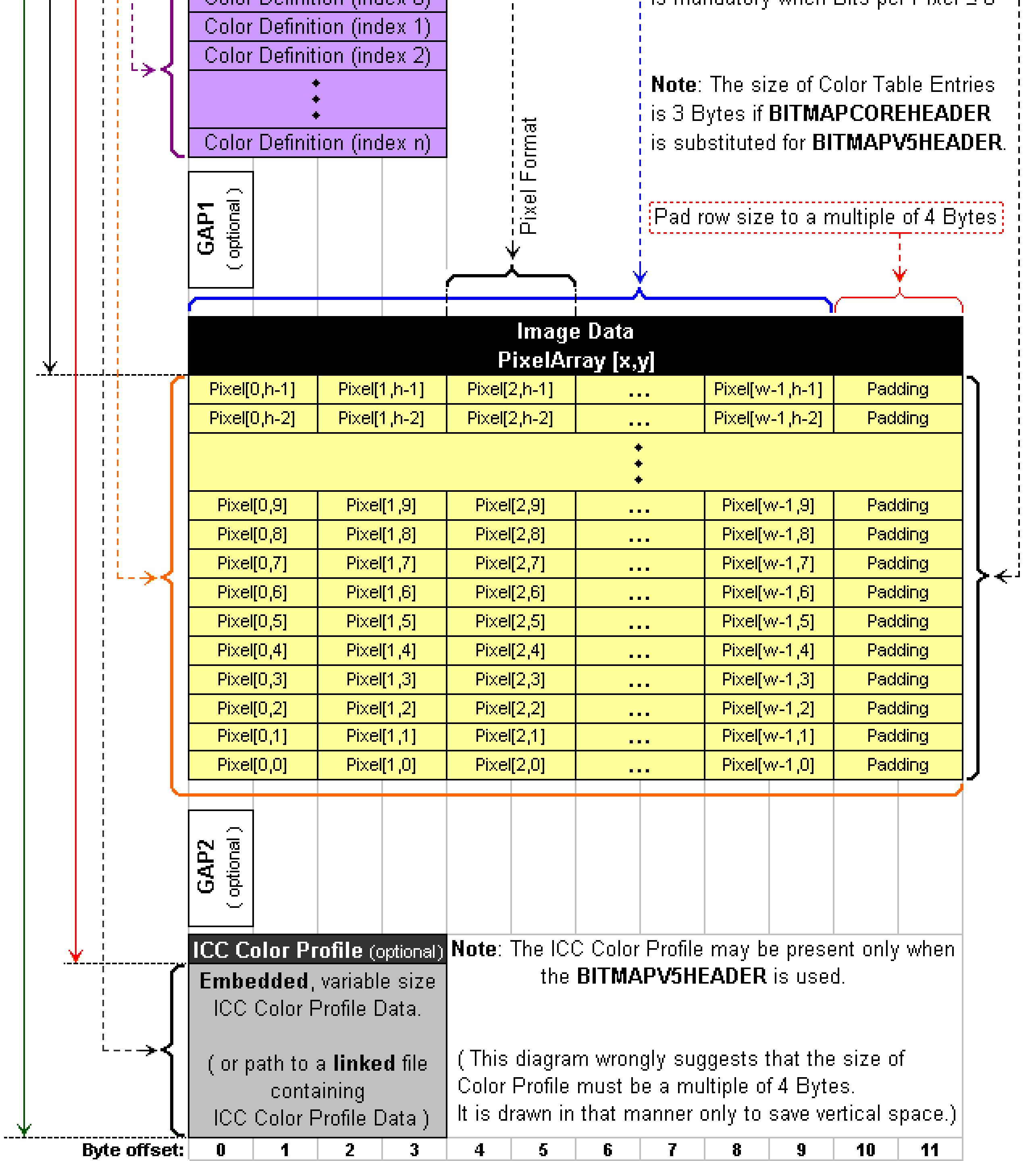
- BMP: "Windows bitmap"
- JPG: Joint Photographic Experts Group
- PNG: Portable Network Graphics
- TIFF: Tagged Image File Format
- GIF: Graphics Interchange Format

- 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)

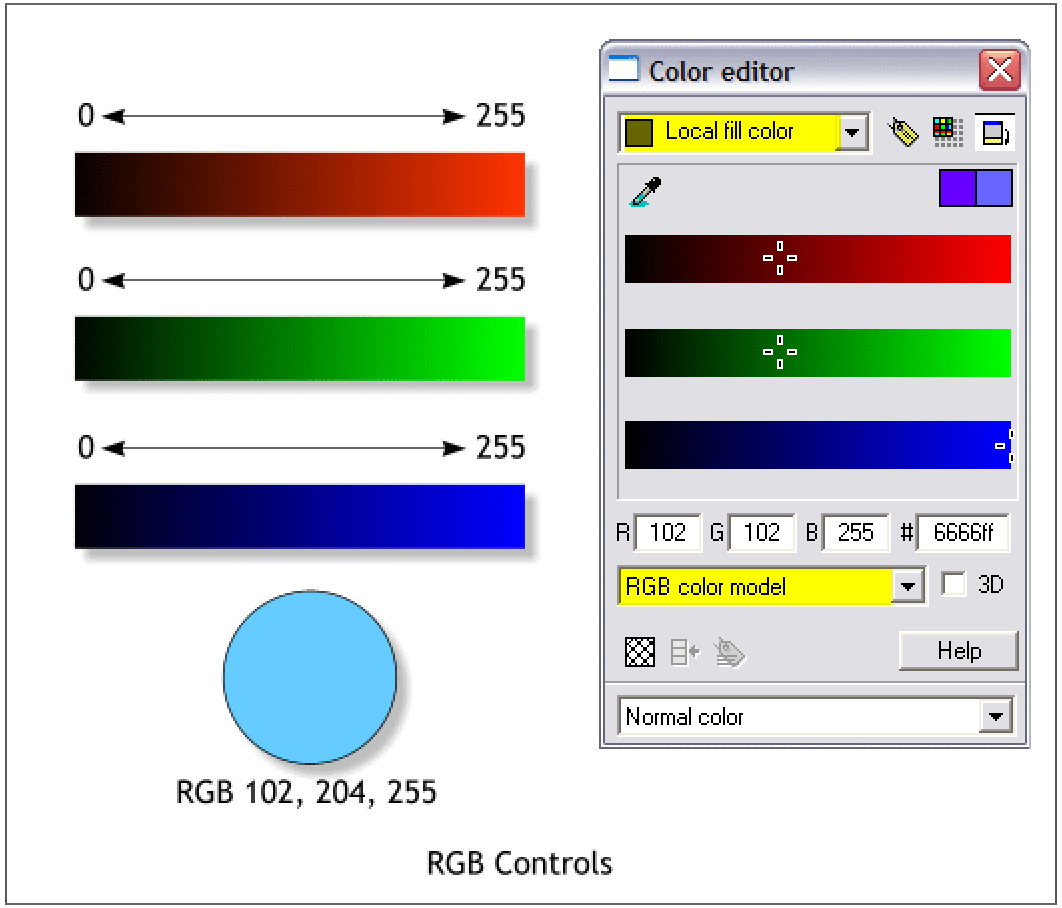
```c
int main(){

  // 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

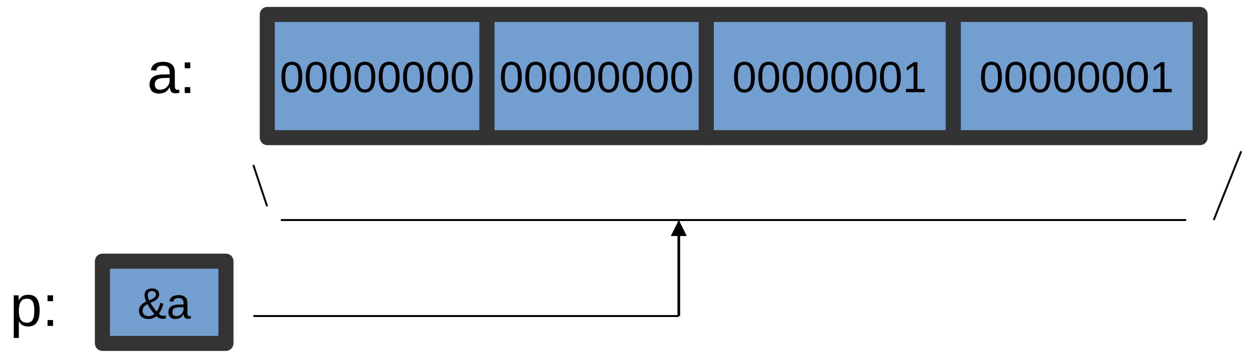
- Other items such as the length and width are 4 byte integers

- It's time to get brave and learn some new tools to work with mixed memory!



Normal pointer use

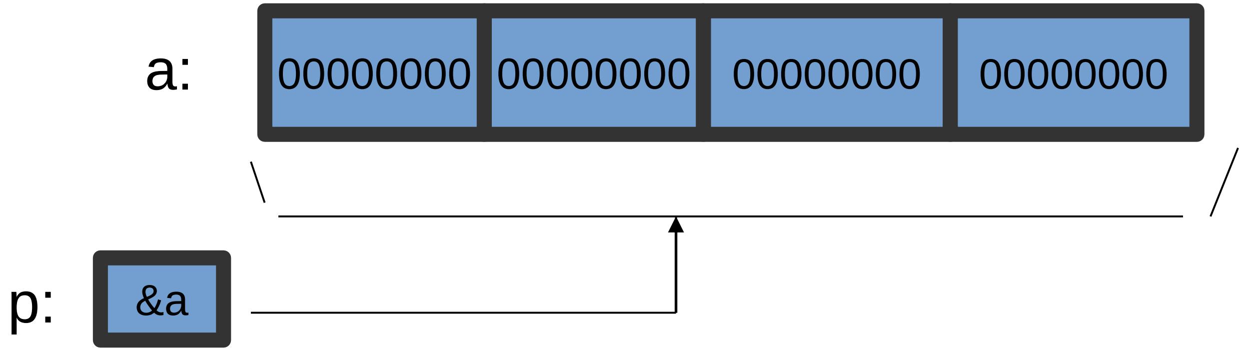
- int *p; -> a pointer to an integer value
- int a = 257; -> an integer variable
- p = &a; -> p now holds the **address of** a (points to a)



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

Pointers can re-inpterpret 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

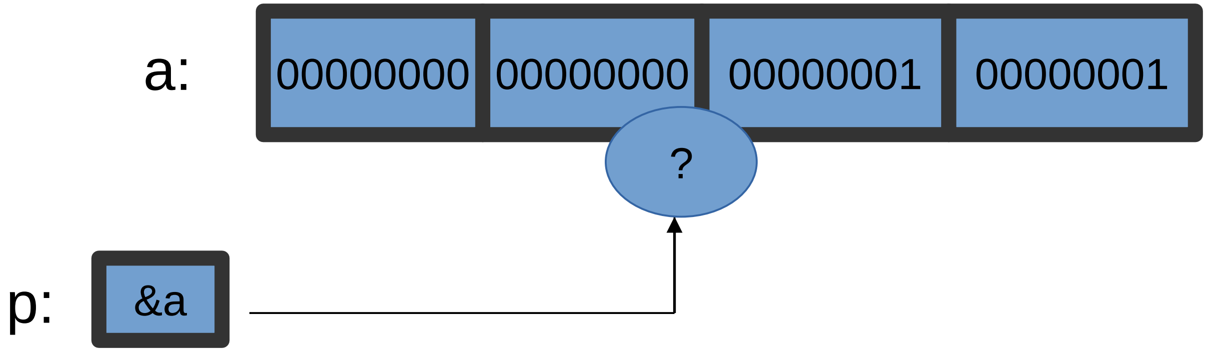
- ExampleCode/Lecture13-ImageFiles/pointer_conversion.c

- 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?

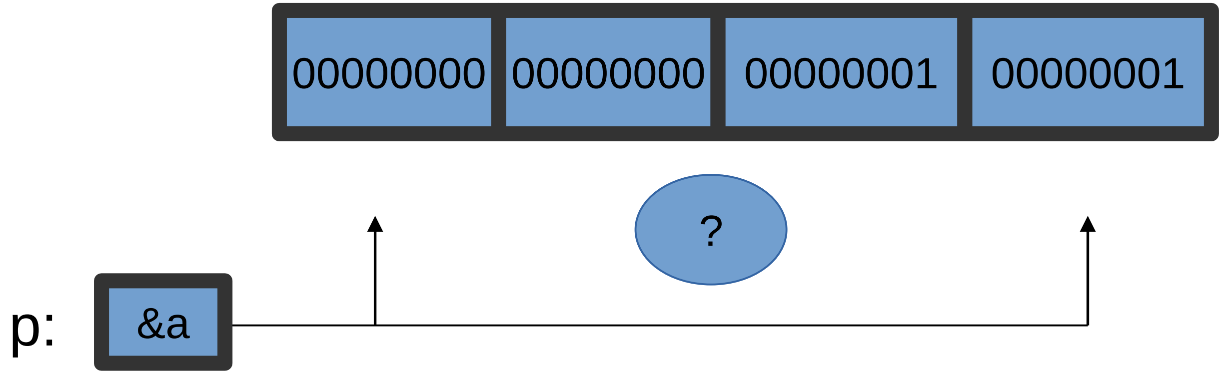
- char *p; -> a pointer to a character value
- int a = 257; -> an integer variable
- p = (char*)&a; -> p now holds the **address of** a,

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



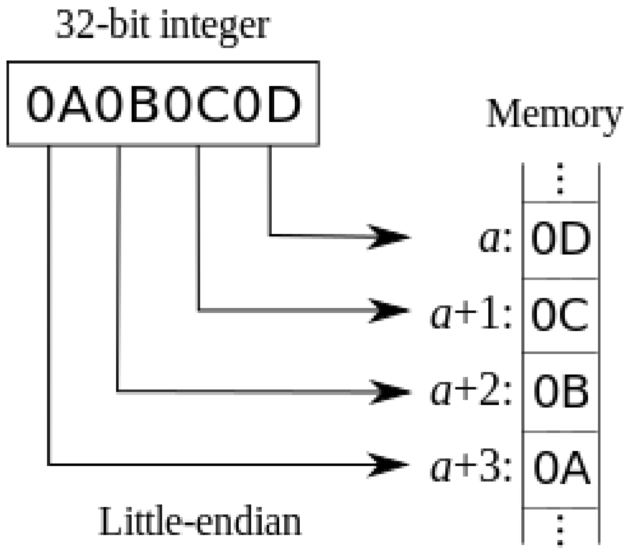

```
int a = 257;
char *p = (char*)&a;
printf( “The value at *p is %d.\n”, *p );
```

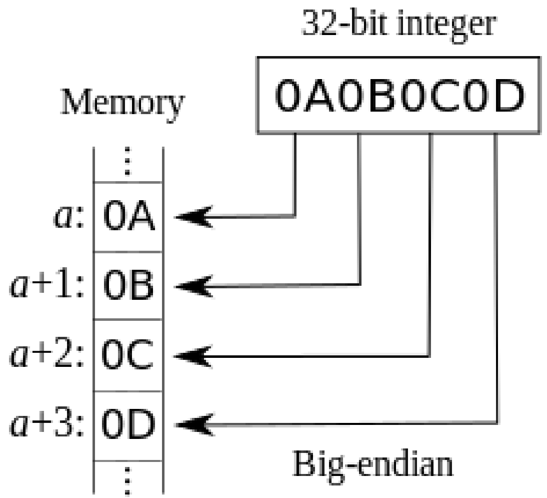
The output of this program depends on “endianess”. Sample code GitHub:

endianness_test.c

Word Endianess

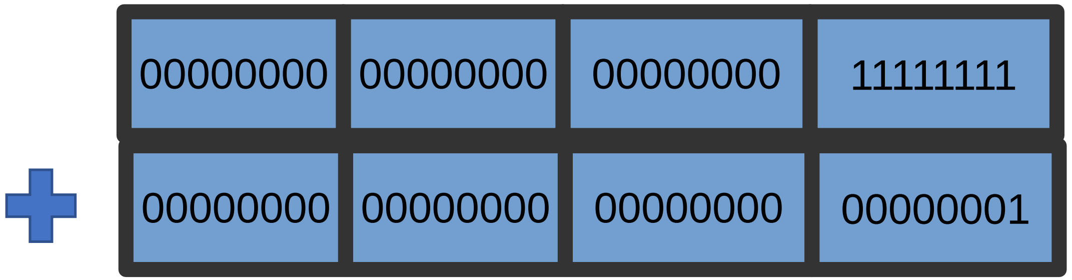
- The order that bytes within an integer are stored in memory is a convention, named Endianess, and there is no right answer.
- Most systems we deal with will be Little-endian, but there are major execptions (the Sun company)
- It is always better to check than to assume



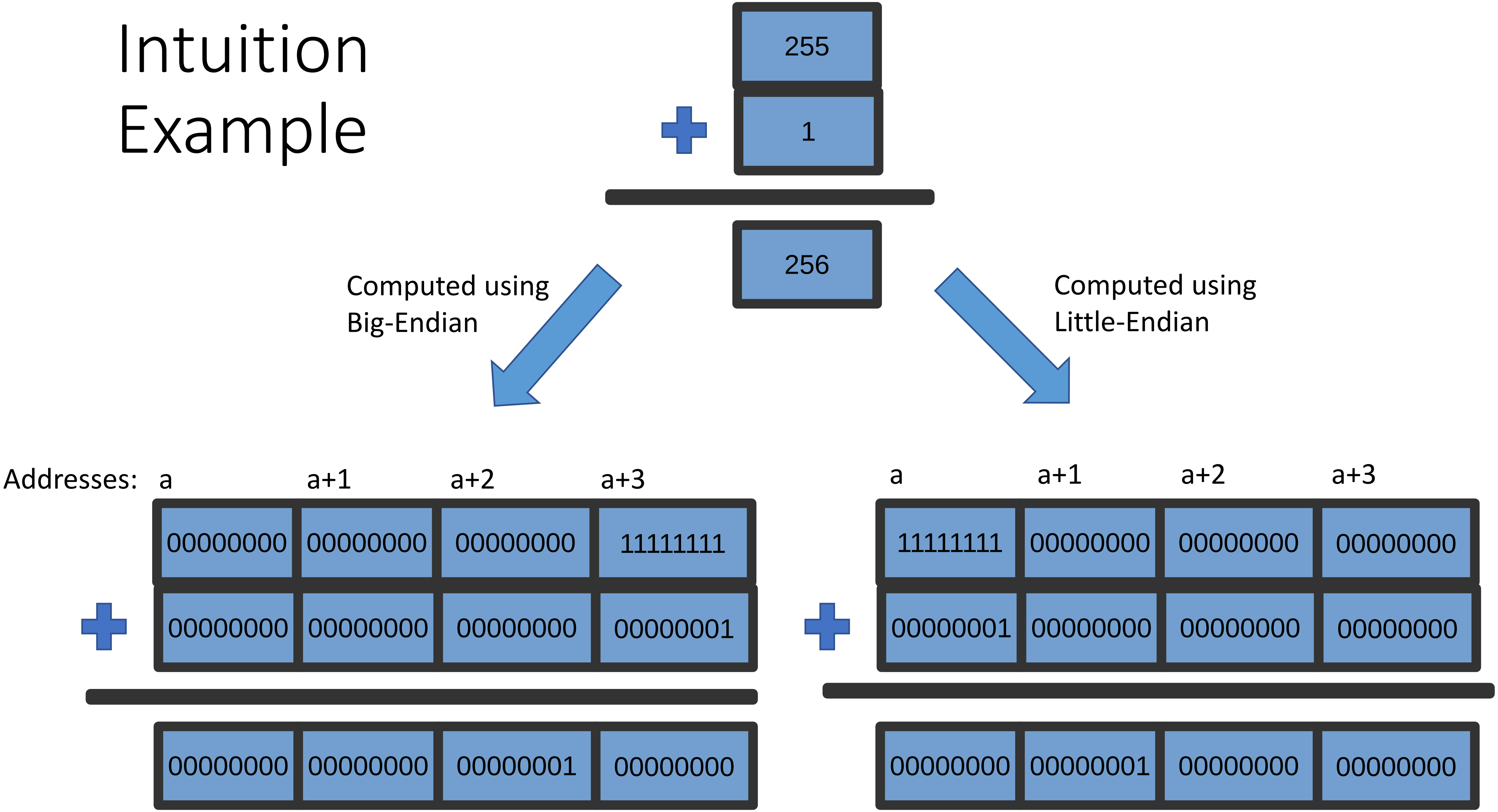


Endianess intuition helper

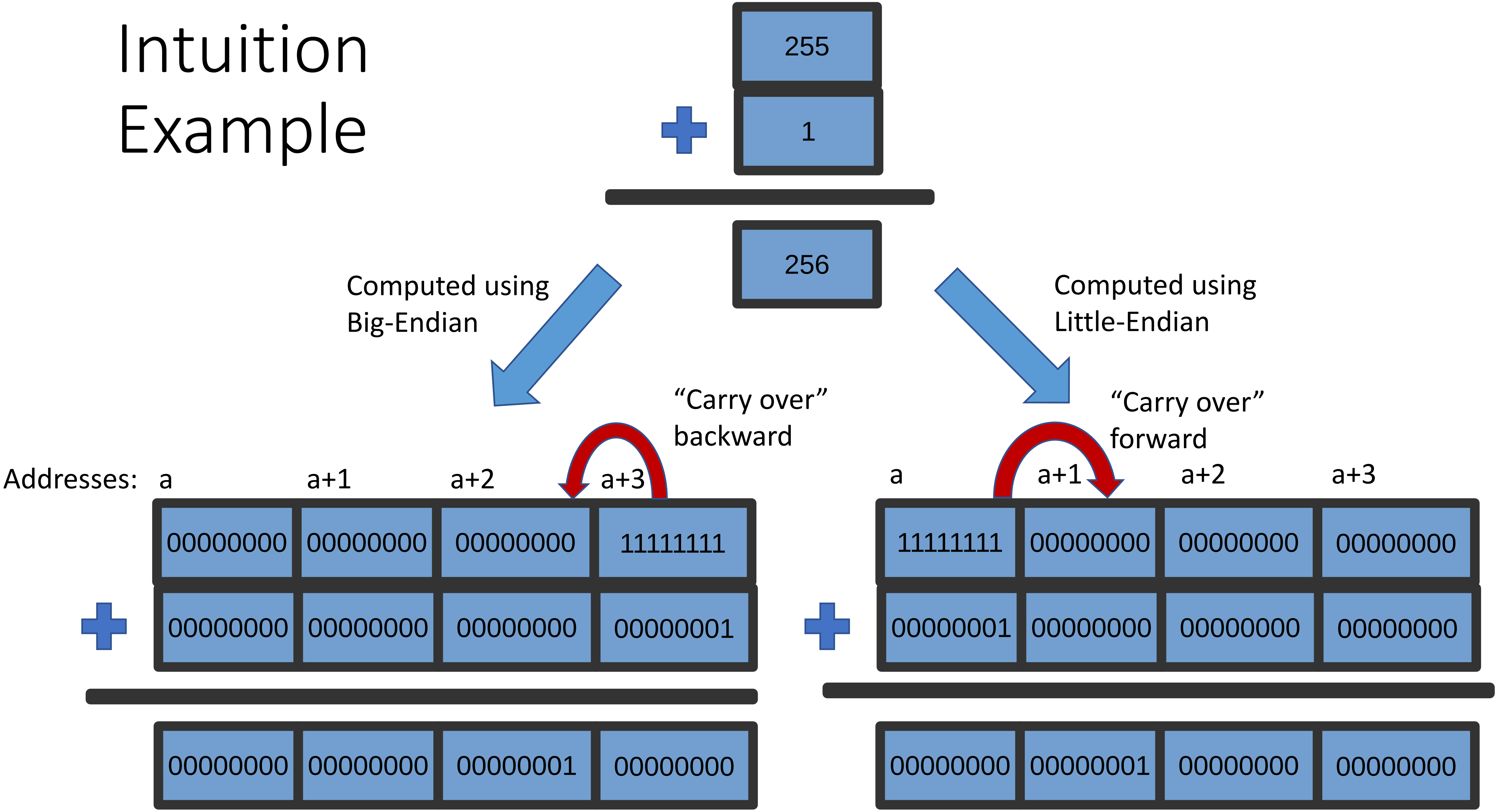
- Little or big? Name determined by the "significance" of the byte at the first address:
  - Little: the least significant comes first
  - Big: the most significant comes first
- Humans always write numbers in Big Endian, why would most computers use Little?
  - Think about doing addition with carry-over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



Intuition Example



Intuition Example



Endianness and Images

- We began to think about this topic because we started to convert between memory of different lengths. Will it really matter for images?

- Not for BMPs, probably. BMP files are explicitly specified to use Little Endian, which is the format of most machines (all x86 compatible):
  - To tell on your machine, try running the sample code or type “uname –a” on the terminal. If you see “x86” anywhere, it’s Little-Endian

- So why does this matter? Because we are not so lucky in many cases:
  - PNG images: https://www.w3.org/TR/PNG/#3byteOrder
  - The Internet mostly uses Big Endian! Hence B.E also called “Network Byte Order”

Reading and Writing Pixels?

- It’s time to get going and change some data. Let’s set all the pixels to white!
  - bmp_2_white.c in ExampleCode

```
// To modify the pixel data, we need to make sure we dont
// mess up the image header. Otherwise, image viewers will
// no longer understand the file. Let's read the offset
// and move a pointer forward.
unsigned int* offsetp = (unsigned int*)(imageData+10);
unsigned int offset = *offsetp;
char *pixel_data = imageData + offset;

// Now, indexing to pixel_data will only change the "visible"
// data in the image. White is all of R, G and B = 255.
// Warning! This changes the "padding" parts of the image also.
// For A3, you will need a more sophisticated way of managing
// the image data.
for( int pixel=0; pixel< overallFileSize-offset; pixel++ )
        pixel_data[pixel] = 255;

// Time to output, just dump the binary data, inverse of reading.
FILE* out_file = fopen( "utah_white.bmp", "wb" );
if( out_file == NULL ){
        printf( "Unable to open utah_white.bmp for writing.\n" );
        return -1;
}
fwrite( imageData, 1, overallFileSize, out_file );
```

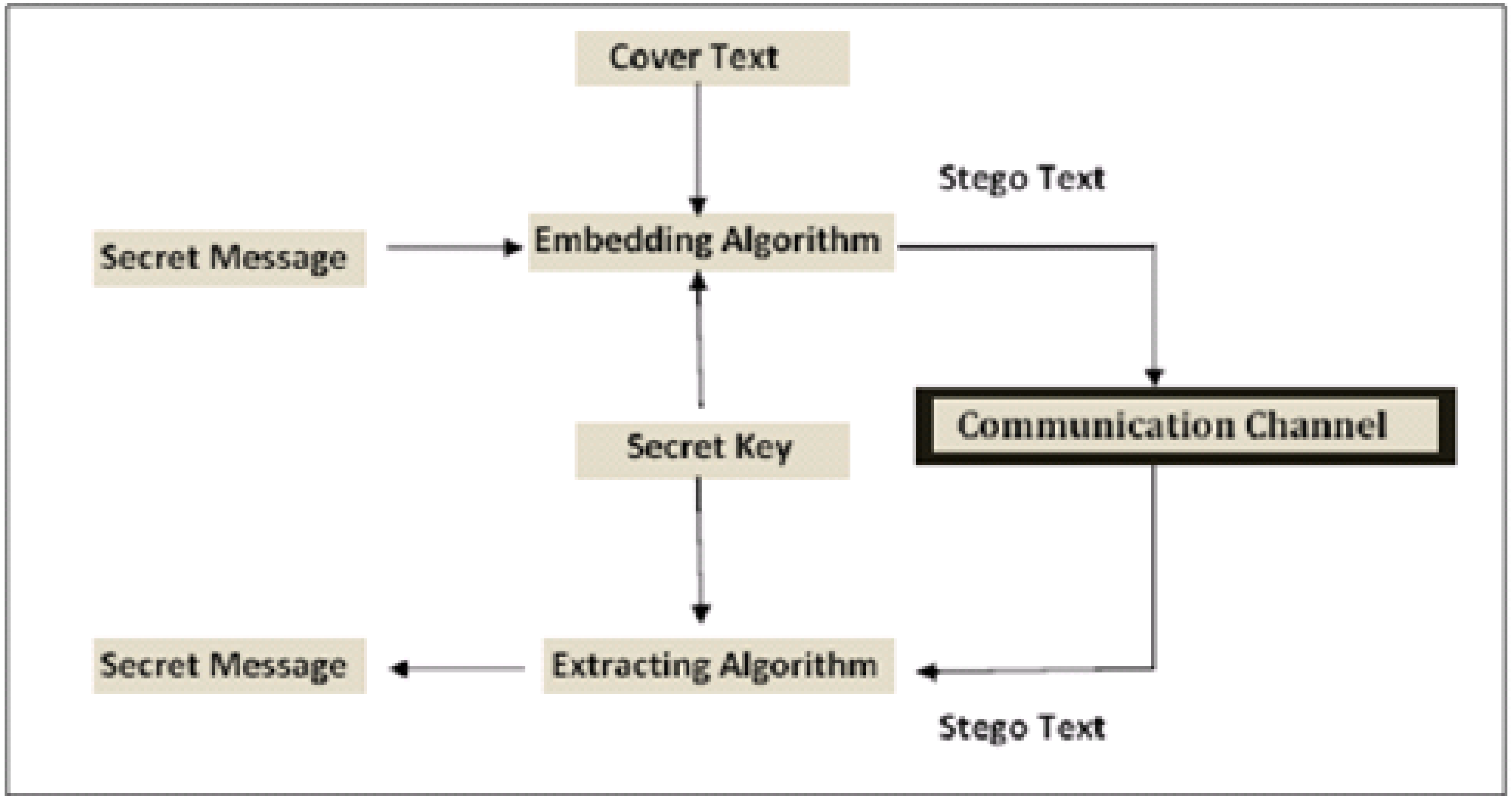
A motivating example: steganography

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

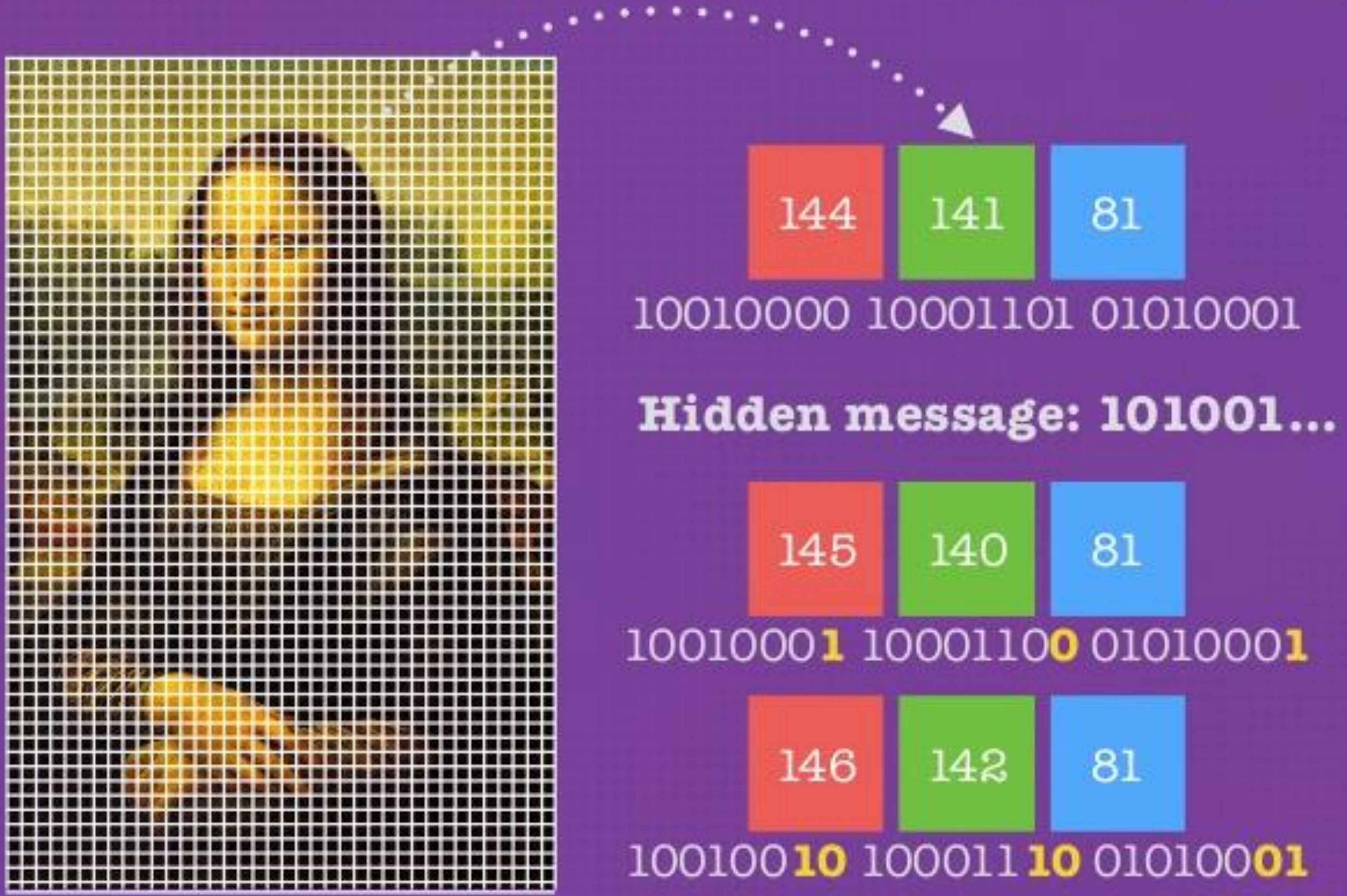
- Two files posted in this ExampleCode/Lecture13 folder on GitHub

- We have not yet covered everything needed to understand them, but we will demo quickly to show what we're after

- The rest of the details in the last part of these slides

Digital Steganography

LSB in Images



Elements Required for In-Image Steganography

- Read and write binary image data (DONE!)
- Read and change pixel values (DONE!)

- Steganography requires us to work with *bits* directly, which we haven’t considered yet:
  - 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:

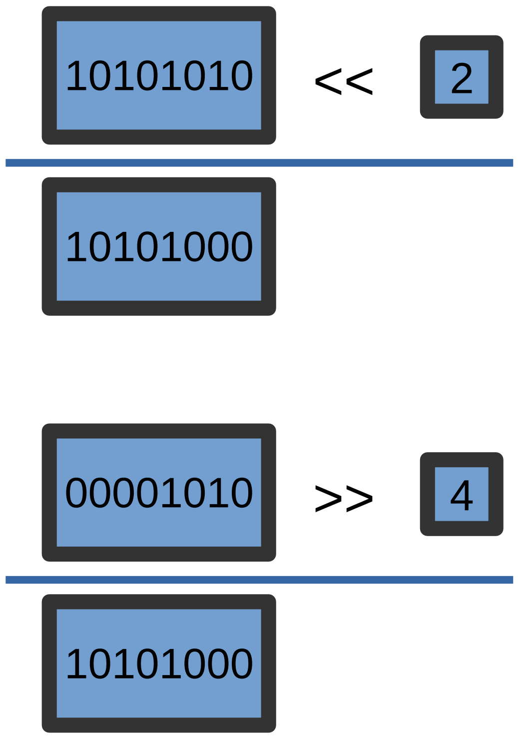
- bit_arg<<shift_arg
  - Shifts bits to of bit_arg shift_arg places to the left -- equivalent to multiplication by 2^shift_arg
- bit_arg>>shift_arg
  - Shifts bits to of bit_arg shift_arg places to the right -- equivalent to integer division by 2^shift_arg

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

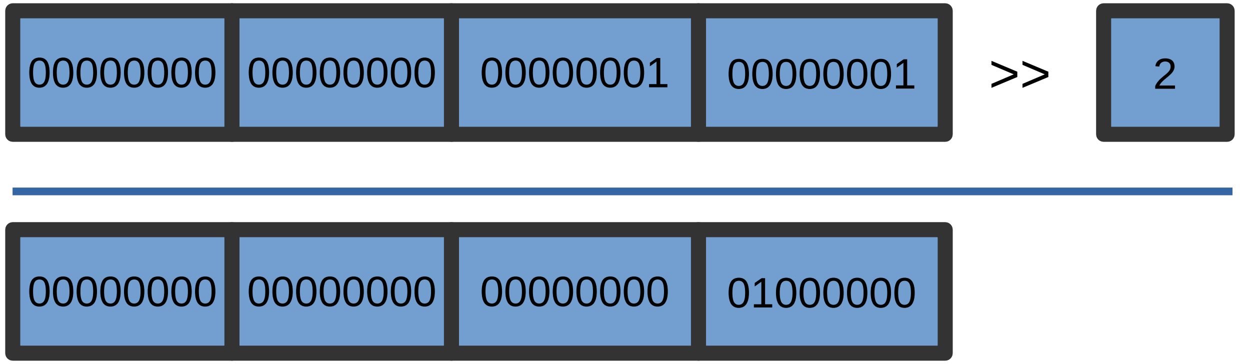
Bit-shift Operators

- Moves the existing bits a specified number of positions left or right.
  - When a bit hits the edge, it is lost
  - New bits always take 0 (for unsigned – we do not cover signed bit shifting in 206)
- Note, shifting is similar to multiplying or dividing by powers of 2



Multi-byte shifting

- Treats the true integer value. That is, we do not have to think about address ordering here. If a bit hits the boundary of its byte, it seamlessly moves to the next using significance order.
- Now, only the least and most significant byte boundaries cause “loss”



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:

	0	0	1	1
&	0	1	0	1
	0	0	0	1

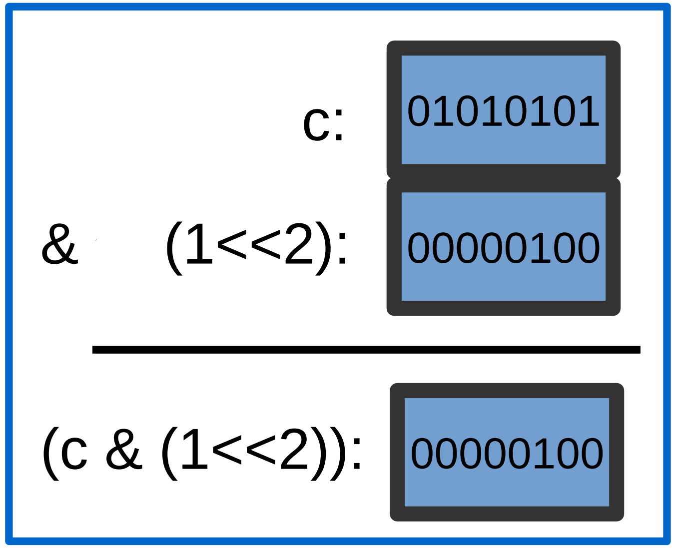
	0	0	1	1
|	0	1	0	1
	0	1	1	1

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

Integrated Bitwise Example

- 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 COMLETE!

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”?

Read More

- K & R textbook Section 2.9