COMP206-BMP Images

BMP: “Windows bitmap”

How do cameras form images?

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

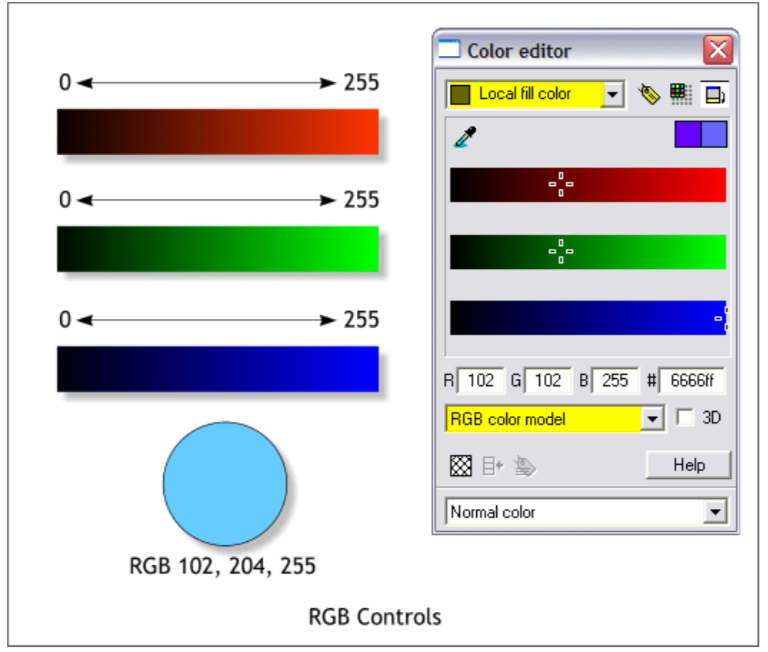
How can we read a BMP file using C?

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 (e.g. \n may be some ramdom color)
* 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 (in image, 0 means black instead of the end)

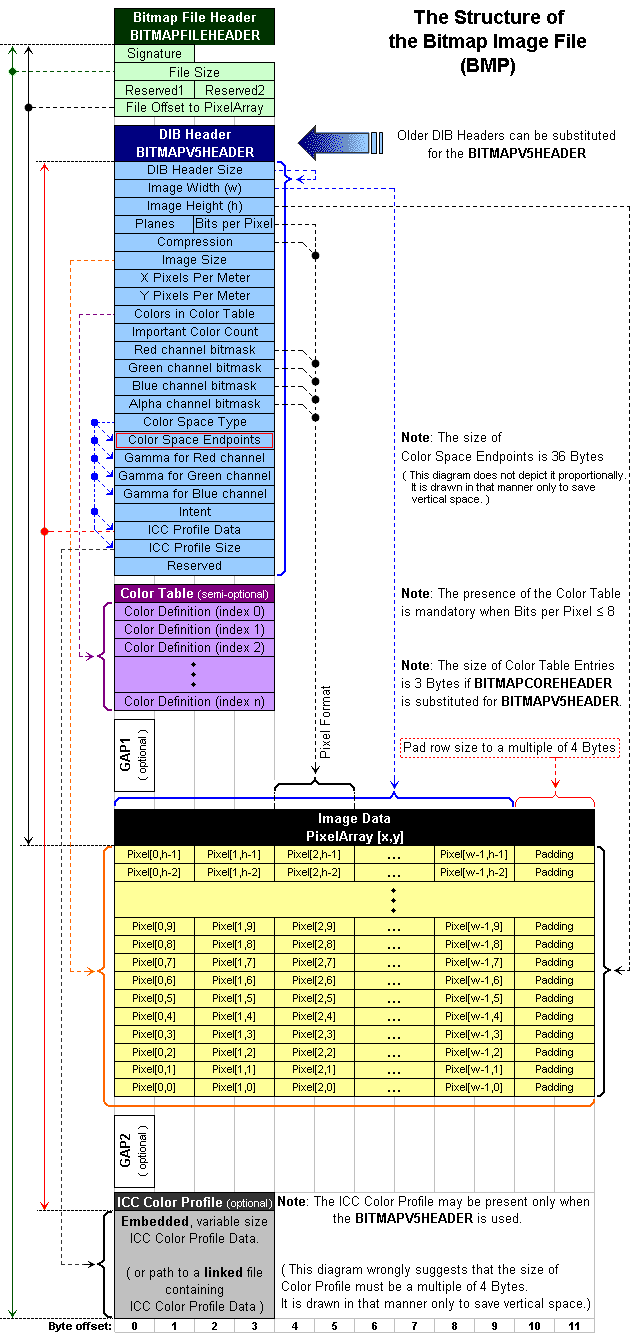


Now we have the data…

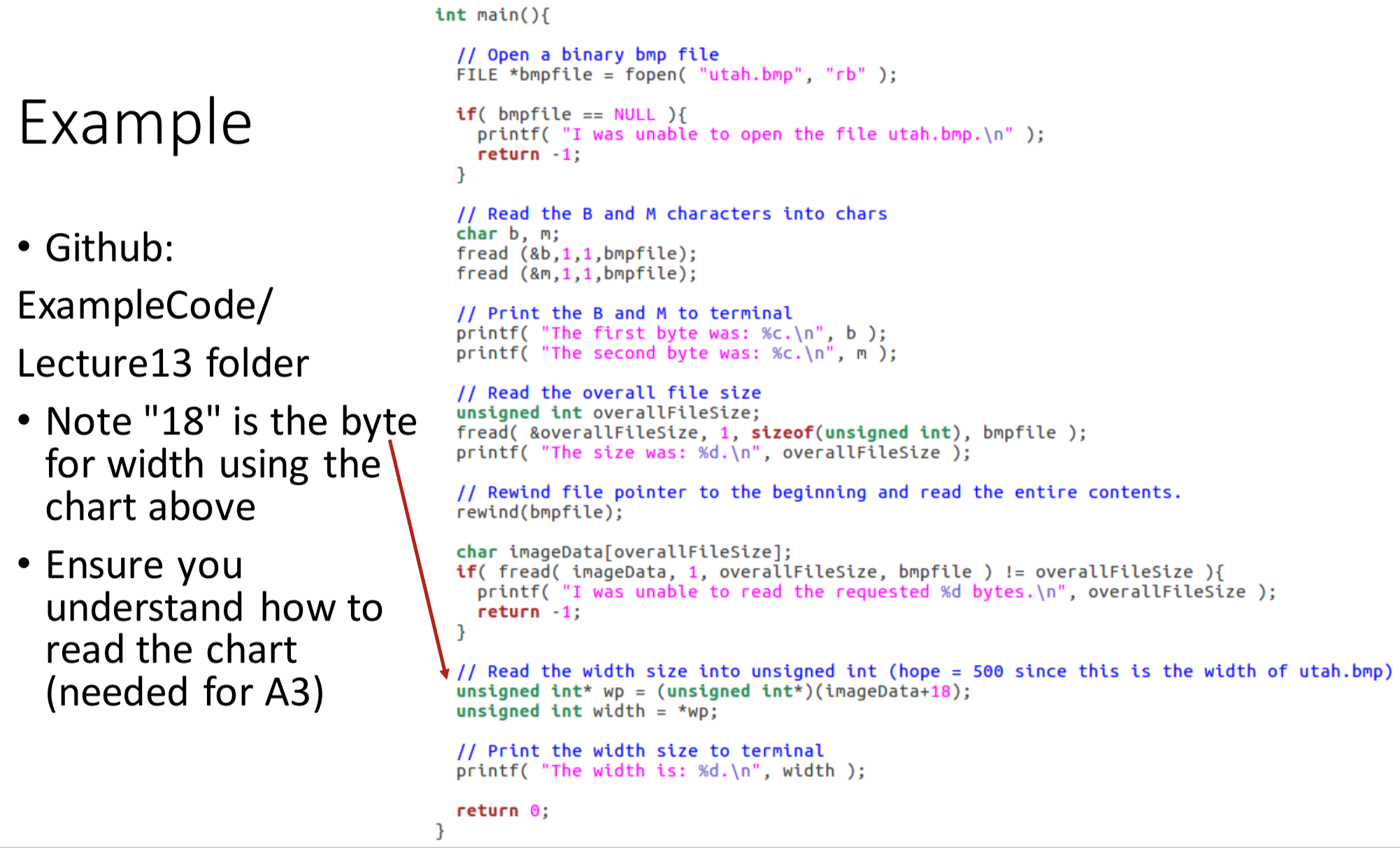
Each color of each pixel is stored as an integer between 0 and 255 (one byte)

Easiest way to work with these in C: represent each as an unsigned char

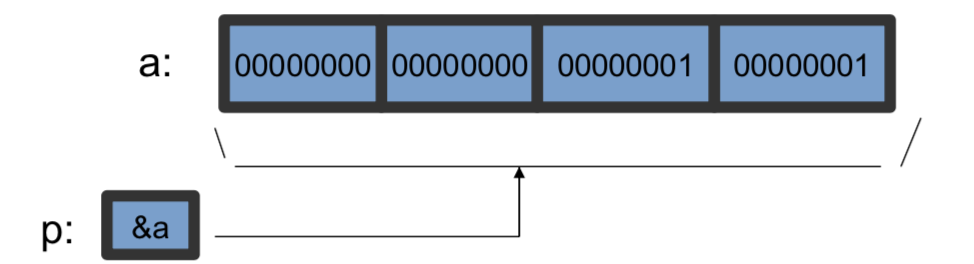
Other items such as the length and width are 4 byte integers



18 bytes



Pointer and Type Conversion

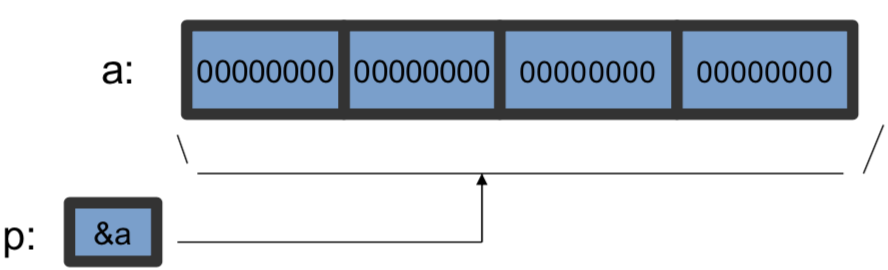


Normal use of a pointer

int \*p;

int a = 257;

p=&a;



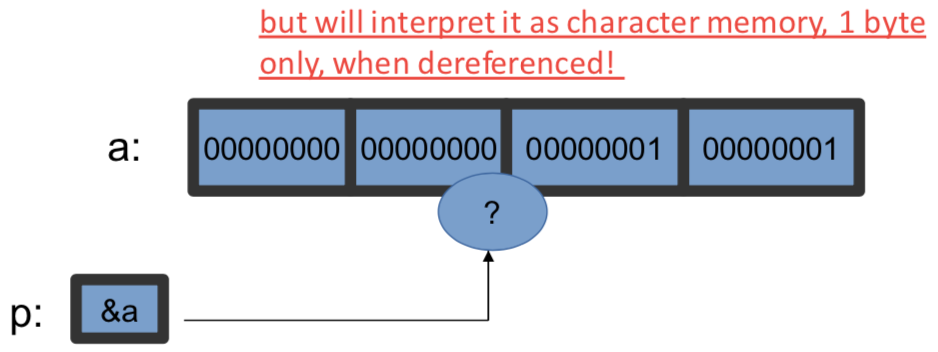
Pointers can re-inpterpret memory

int\*p;

char a[4];

memset( a, '\0', 4 );

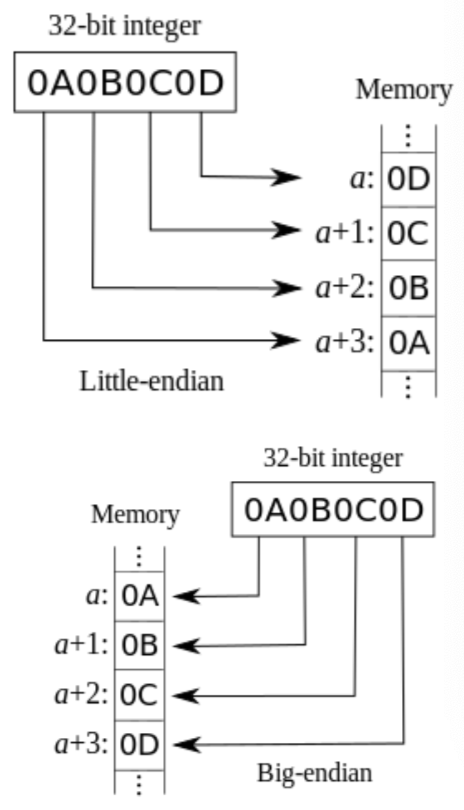
p = (int\*)a;



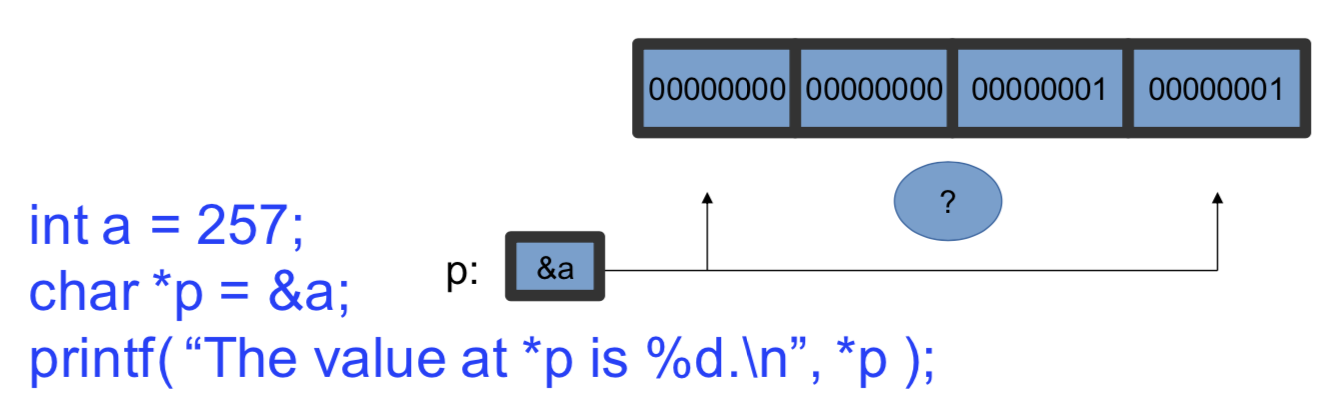
char \*p;

int a = 257;

p = (char\*)&a;

Word Endianness

The output of this program depends on “endianness”



*The order that bytes within an integer are stored in memory is a convention, named Endianness*, 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

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"

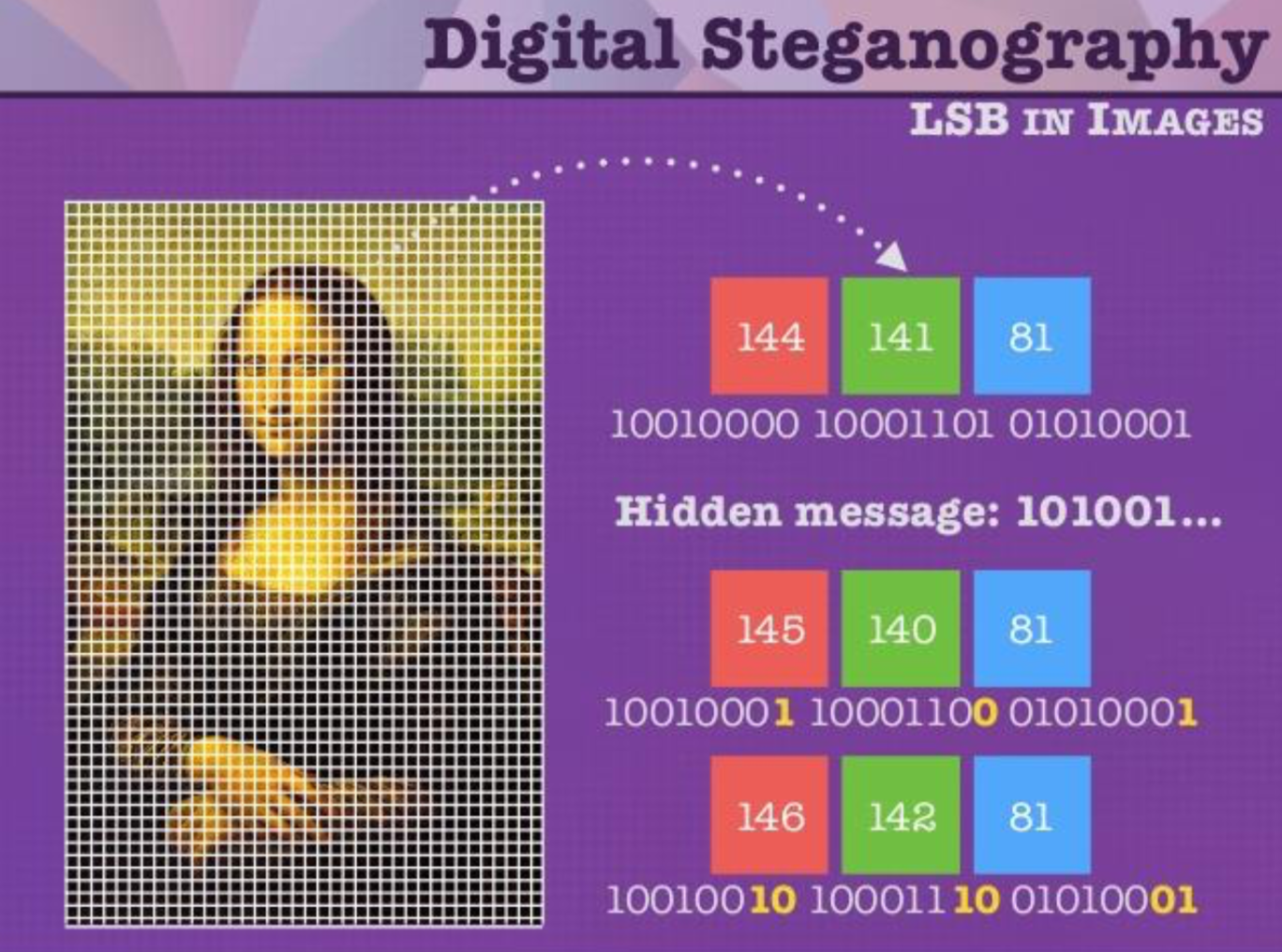
Steganography

*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.

[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

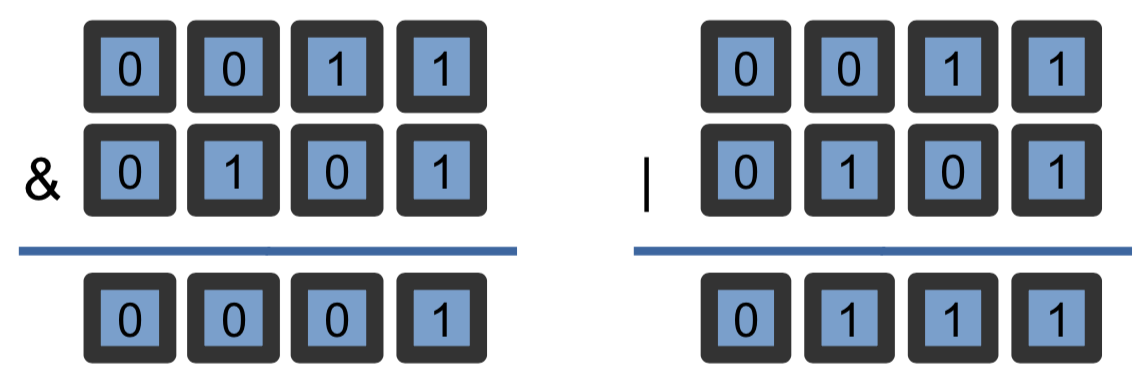
− Takes the bitwise XOR of left\_arg and right\_arg (one or the other but not both)

* ~arg

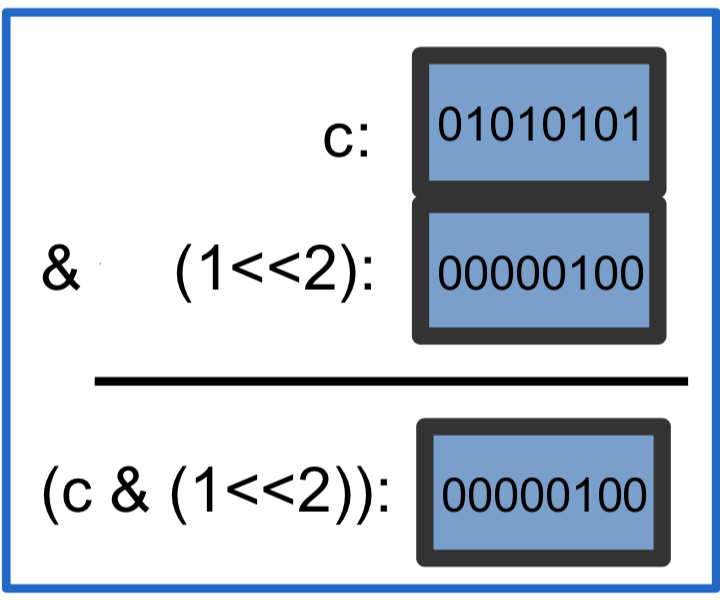
− Takes the bitwise complement of arg

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:



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



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” );

Set the value of bit 8 to1

10000000=128 or 1<<7, use |

Count the number of 1’s

Find the first 0 starting from the right

leave things alone

& 1 | 0