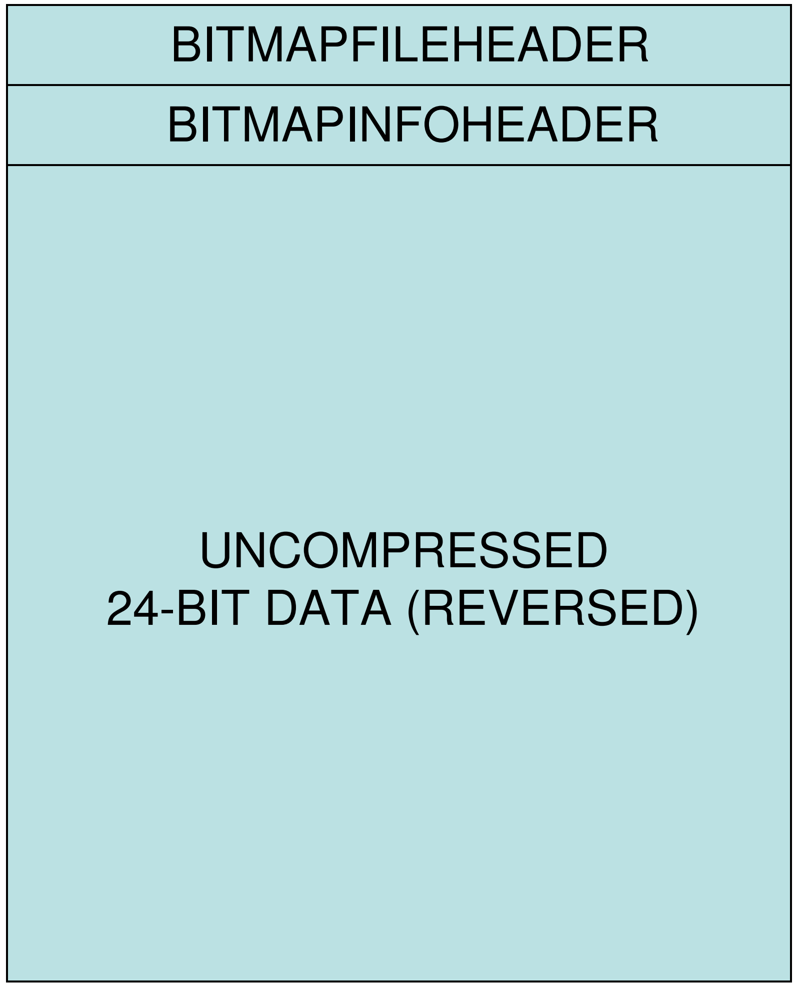
Intro to C++

Lecture 9

BMP Files, Bitwise Operators, RSA Encryption

BMP Structure



Steps for Loading

1. Load the two bitmap headers.
2. Dynamically allocate enough memory to hold the bitmap.
3. Load the bitmap data (usually reversed) into the newly allocated memory.

Header Structure

```
typedef struct tagBITMAPFILEHEADER {
        WORD    bfType;         // if not 'MB', is NOT a BMP!
        DWORD   bfSize;
        WORD    bfReserved1;
        WORD    bfReserved2;
        DWORD   bfOffBits;      // byte offset of data from beginning
} BITMAPFILEHEADER, FAR *LPBITMAPFILEHEADER, *PBITMAPFILEHEADER;
```

```
BITMAPFILEHEADER bitmapHeader;
inFile.read ( (char *)&bitmapHeader,  sizeof (BITMAPFILEHEADER));
```

NOTE: 4-byte alignment required!

Info Structure

```
typedef struct tagBITMAPINFOHEADER{
        DWORD       biSize;
        LONG        biWidth;            // Width
        LONG        biHeight;           // Height
        WORD        biPlanes;
        WORD        biBitCount;         // Bits per pixel (should be 24 or 32)
        DWORD       biCompression;      // Should be BI_RGB
        DWORD       biSizeImage;
        LONG        biXPelsPerMeter;
        LONG        biYPelsPerMeter;
        DWORD       biClrUsed;
        DWORD       biClrImportant;
} BITMAPINFOHEADER, FAR *LPBITMAPINFOHEADER, *PBITMAPINFOHEADER;
```

```
BITMAPINFOHEADER bitmapInfoHeader;
inFile.read ( (char *)&bitmapInfoHeader,  sizeof (BITMAPINFOHEADER));
```

BMP Data

- The BMP data begins at `bfOffBits` **bytes** from the beginning of the file.
- Each BMP row is ***padded*** so that it is a multiple of 4 bytes, so read the bitmap row-by-row.
- Note that if `biBitCount` == 24 then each color will be represented by **three** bytes.
- If the `biHeight` is positive, then the image's rows will be reversed – the first row will correspond to the last row in your

Wotsit?

You can find more on the BMP file format (and just about any other file format) by visiting the below website:

www.wotsit.org

Bitwise Operators

- **Bitwise operators** are operators that perform direct logic operators on individual bits.
- We’ve already seen examples of these – the left and right bit-shift operators!

Bitshifts

- ‘<<‘ shifts the bits in a variable to the *left*.
- ‘>>’ shifts the bits in a variable to the *right*.
- Both operators insert zeros and remove ones.
- Examples:

```
unsigned char myVar = 34;    // 00100010_2
myVar = myVar << 2;          // 10001000_2
myVar <<= myVar;             // 01000000_2
myVar = myVar >> 3;          // 00001000_2
```

Bitshift Trick

- Note that the bitshift is an incredibly fast way to multiply or divide by powers of two! This method used to be magnitudes faster than the equivalent multiplication or division:

```
unsigned char myVar = 34;    // 00100010₂ (34)
myVar = myVar << 2;          // 10001000₂ (34 * 2^2 = 136)
myVar <<= 3;                 // 01000000₂ ((136 * 2^3)%256 = 64)
myVar = myVar >> 3;          // 00001000₂ (64 / 2^3 = 8)
```

Bitwise Operators

- Boolean operators can be applied to bits. From these operators, we can derive most traditional operations such as addition and division.

```
NOT (~): ~0 = 1, ~1 = 0.

AND (&): 0 & 0 = 0, 0 & 1 = 0, 1 & 0 = 0, 1 & 1 = 1.
OR  (|): 0 | 0 = 0, 0 | 1 = 1, 1 | 0 = 1, 1 | 1 = 1.
XOR (^): 0 ^ 0 = 0, 0 ^ 1 = 1, 1 ^ 0 = 1, 1 ^ 1 = 0.
```

Bitwise Operators

- More complex examples:

```
  11001100     11001100      11001100   ~ 11001010
& 01010101   | 01010101    ^ 01010101   ----------
----------   ----------    ----------     00110101
  01000100     11011101      10011001
```

Bitmasks

- How can we pack several flags into a single variable, for instance to send as parameters to a function?

```
#define MB_OK               1   // 00000001₂
#define MB_OKCANCEL         2   // 00000010₂
#define MB_YESNO            4   // 00000100₂
#define MB_YESNOCANCEL      8   // 00001000₂

#define MB_ICONHAND         16  // 00010000₂
#define MB_ICONQUESTION     32  // 00100000₂
#define MB_ICONEXCLAMATION  64  // 01000000₂

#define MB_NOFOCUS          128 // 10000000₂
```

mbParam = MB_OK | MB_ICONHAND = 00010001_2

mbParam & MB_OK = 1

mbParam & MB_ICONHAND = 1

mbParam & MB_YESNO = 0

RSA Cryptography

- RSA – Rivest, Shamir, and Adleman, three professors who discovered this means of encryption.
- RSA relies on the fact that it is easy to multiply two large prime numbers, but it's very difficult to factor the product.
- Other easy one way yet hard the other mathematical techniques exist such as elliptical curves.

RSA Cryptography

- We initially enter the ***plaintext***, which is typically an unencrypted text string.
- The algorithm works, and it returns ***cyphertext***, the encrypted plaintext.
- Using RSA, keys are needed to create and decypher the *cyphertext*. ***Public keys*** are accessible by everyone and are used to encode *plaintext*. ***Private keys*** are required to decode *plaintext* encrypted with a certain *public key*.

RSA Cryptography

- Computer scientists typically use Alice and Bob (and occasionally more) to describe the ones transmitting the message, and Eve as the eavesdropper trying to read or alter the *plaintext*.
- Let's go through a sample encryption.

Sample Encryption

- Alice wishes to send a message to Bob.
- Bob picks two prime numbers and finds their product:

`P = 37, Q = 17`

`PQ = 629`

- Bob now gives the product to Alice.
- Bob also gives a second number which has no common factors with (P-1)(Q-1) = 576 (we'll use E=19).

Sample Encryption

- Alice first changes her message ("ACE") into numbers (A = 1, B = 2, … Z = 26), so ACE = 135.
- To translate this into ***cyphertext***, Alice performs a simple calculation:

```
M^E mod PQ = 135^19 mod 629
           =   50
```

Sample Encryption

- Now Bob can use the following formula to find the original ***plaintext.*** Choose a value X such that D is an integer:

```
d = (X(P-1)(Q-1) + 1) / E
  = (X(576) + 1) / 19
  = 91 (when X = 3)
```

- And now we can find the original *plaintext*:

```
(M^E)^d mod PQ = (50)^91 mod 629
               = 135
```

Implications

- Note that Bob sent Alice a ***public key***, PQ and his number E. Bob retains the ***private key*** necessary to decode the text, P and Q.
- Note that only Bob knows (P-1)(Q-1), so the algorithm's security rests on the fact that it is difficult to factor PQ.
- In reality, the prime numbers chosen typically are hundreds of digits long (e.g. 128- or 256-bit encryption)
- To practically use this algorithm, we must have a good way of exchanging keys (this is not specified by RSA). The first popular exchange was the ***Diffie-Hellman key exchange***.
- Try encrypting some numbers or even writing your own encryption software!
- And this is what started the E-Commerce revolution!

TODO

- Download ITCC_HW4.zip from the site (will be available soon) http://www.pclx.com/itcc/, and complete the homework exercises, emailing them (FOR THIS WEEK ONLY) to itcc_teachers@pclx.com. Please do not resubmit solutions, even if they are revised. All homework must be submitted by 6:00am PST August 5.
- This is the second-to-last assignment! Lectures will end Thursday, August 5!
- If you finish with the homework, experiment!