

CSE 100: STREAM I/O, BITWISE --- OPERATIONS, BIT STREAM I/O

PA 2 Implementation strategy

- Implement Huffman tree `build()` method
 - `HCTree.cpp` and `HCTree.h`
- Write verification code to check that you can construct simple Huffman trees correctly
 - Use small inputs that you can verify by hand
 - Output codes as strings of 1s and 0s (char)
- Write the encoder method and driver program
 - Test with simple inputs that you can verify by hand and output the encoded input as character strings of 1s and 0s
- Add binary I/O
 - Write implementations of `BitInputStream` and `BitOutputStream` that write/read the compressed file as a text file Compress/decompress a small file (100 bytes)
- Decompression should map the encoded input back to the original input

Today's Class

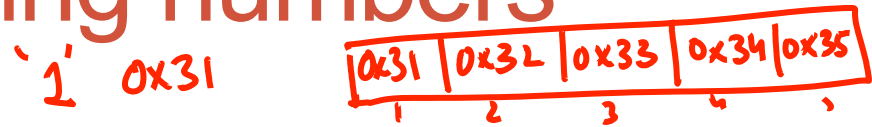
- C++ I/O
- I/O buffering
- Bit-by-bit I/O

Reading and writing numbers

```
#include <iostream>
#include <fstream>
```

```
using namespace std;
```

```
int main( int argc, char** argv )
{
    ofstream numFile;
    int num = 12345;
    numFile.open( "numfile" );
    numFile << num;
    numFile.close();
}
```



num

0x3039

4 byte representation of num
in memory

Assuming ints are represented with 4 bytes, how large is numfile after this program runs?

- A. 1 byte
- B. 4 bytes
- ☒ C. 5 bytes
- D. 20 bytes

Reading and writing numbers

```
#include <iostream>
#include <fstream>

using namespace std;

int main( int argc, char** argv )
{
    ofstream numFile;
    int num = 12345;
    numFile.open( "numfile" );
    numFile << num;
    numFile.close();
}
```

You'll need to include delimiters between the numbers

num 0x00003039

Writing raw numbers

headump numfile

Big Endian 0x00 0x00 0x30 0x39
Little Endian 0x39 0x30 0x00 0x00

```
#include <iostream>
#include <fstream>
```

```
using namespace std;
```

```
int main( int argc, char** argv )
{
    ofstream numFile;
    int num = 12345;
    numFile.open( "numfile" );
    numFile.write( (char*)&num, sizeof(num) );
    numFile.close();
}
```

This is the method you'll use for the final submission

Reading raw numbers

```
#include <iostream>
#include <fstream>

using namespace std;

int main( int argc, char** argv )
{
    ofstream numFile;
    int num = 12345;
    numFile.open( "numfile" );
    numFile.write( (char*)&num, sizeof(num) ) ;
    numFile.close();

    // Getting the number back!
    ifstream numFileIn;
    numFileIn.open( "numfile" );
    int readN;
    numFileIn.read( (char*)&readN, sizeof(readN) );
    cout << readN << endl;    12345
    numFileIn.close();
}
```

Opening a binary file

```
#include <iostream>
#include <fstream>

using namespace std;

int main( int argc, char** argv )
{
    ifstream theFile;
    unsigned char nextChar;
    theFile.open( "testerFile", ios::binary );
    while ( 1 ) {
        nextChar = theFile.get();
        if (theFile.eof()) break;
        cout << nextChar;

    }
    theFile.close();
}
```



Binary and nonbinary file streams

- Ultimately, all streams are sequences of bytes: input streams, output streams... text streams, multimedia streams, TCP/IP socket streams...
- However, for some purposes, on some operating systems, text files are handled differently from binary files
 - Line termination characters for a particular platform may be inserted or removed automatically
 - Conversion to or from a Unicode encoding scheme might be performed
- If you don't want those extra manipulations to occur, use the flag `ios::binary` when you open it, to specify that the file stream is a binary stream
- To test your implementation on small strings, use formatted I/O
- Then add the binary I/O capability
 - But there is one small detail: binary I/O operates on units of information such as whole bytes, or a string of bytes
 - We need variable strings of bits

Reading binary data from a file: an example

```

#include <fstream>
using namespace std;
/** Count and output the number of times char 'a' occurs in
    * a file named by the first command line argument. */
int main(int argc, char** argv) {
    ifstream in;
    in.open(argv[1], ios::binary);
    int count = 0;
    unsigned char ch;
    while(1) {
        ch = in.get(); // or: in.read(&ch,1);
        if(! in.good() ) break;    // failure, or eof
        if(ch == 'a') count++;    // read an 'a', count it
    }

    if(! in.eof() ) { // loop stopped for some bad reason...
        cerr << "There was a problem, sorry." << endl;
        return -1;
    }
    cerr << "There were " << count << " 'a' chars." << endl;
    return 0;
}

```

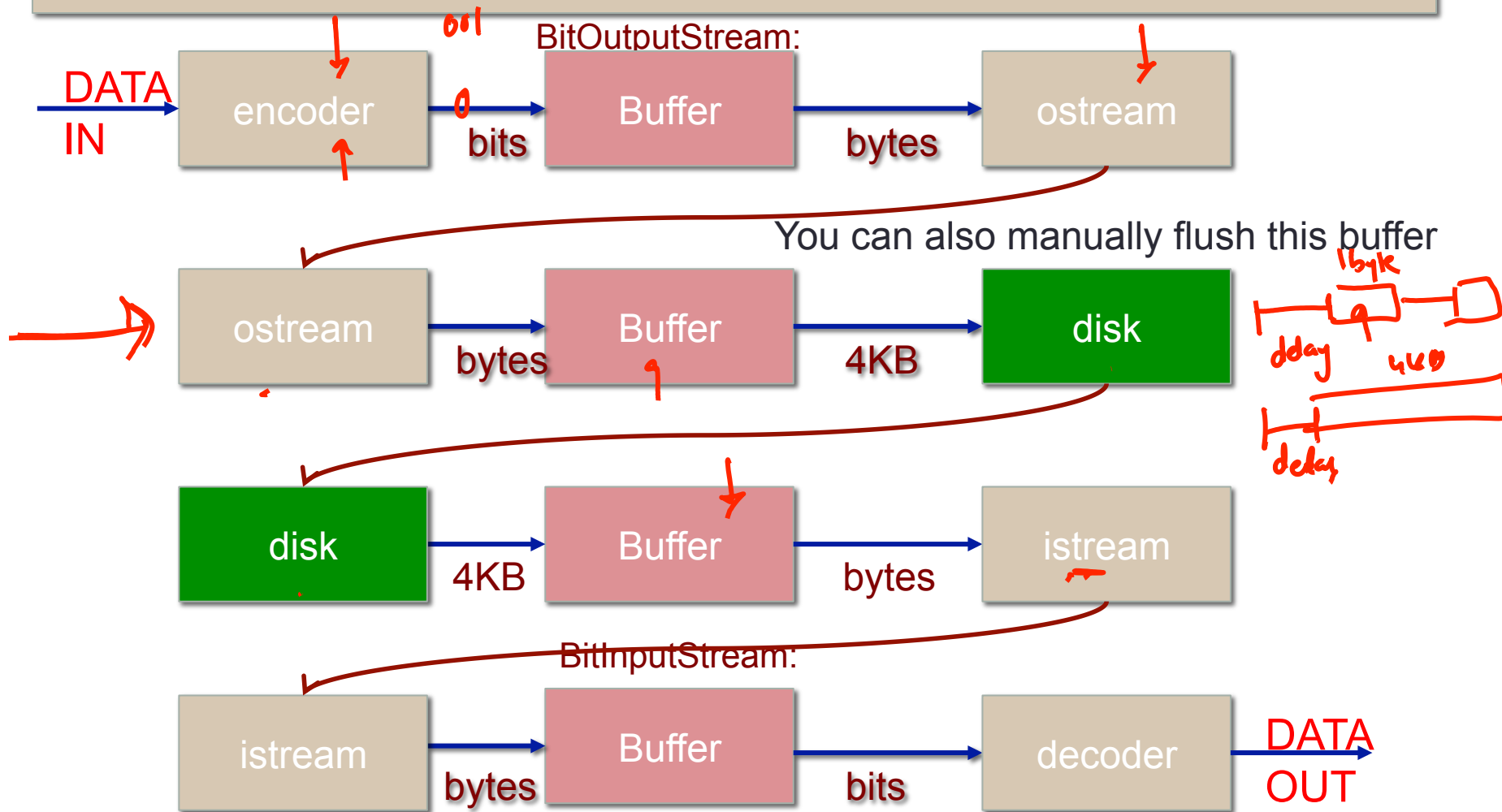
Today's Class

- C++ I/O
- I/O buffering
- Bit-by-bit I/O

Buffering

- The C++ I/O classes `ofstream`, `ifstream`, and `fstream` use *buffering*
- I/O buffering is the use of an intermediate data structure, called the buffer, usually an array used with FIFO behavior, to hold data items
 - Output buffering: the buffer holds items destined for output until there are enough of them to send to the destination; then they are sent in one large chunk
 - Input buffering: the buffer holds items that have been received from the source in one large chunk, until the user needs them
- The reason for buffering is that it is often much faster per byte to receive data from a source, or to send data to a destination, in large chunks, instead of one byte at a time
- This is true, for example, of disk files and internet sockets; even small buffers (512 or 1K bytes), can make a big difference in performance
- Also, operating system I/O calls and disk drives themselves typically perform buffering

Streams and Buffers



Buffering and bit-by-bit I/O

- The standard C++ I/O classes do not have any methods for doing I/O a bit at a time
- The smallest unit of input or output is one *byte* (8 bits)
- This is standard not only in C++, but in just about every other language in the world
- If you want to do bit-by-bit I/O, you need to write your own methods for it
- Basic idea: use a byte as an 8-bit buffer!
 - Use bitwise shift and or operators to write individual bits into the byte, or read individual bits from it;
 - flush the byte when it is full, or done with I/O
- For a nice object-oriented design, you can define a class that extends an existing iostream class, or that delegates to an object of an existing iostream class, and adds `writeBit` or `readBit` methods (and a `flush` method which flushes the 8-bit buffer)

Today's Class

- C++ I/O
- I/O buffering
- **Bit-by-bit I/O**

C++ bitwise operators

- C++ has bitwise logical operators `&`, `|`, `^`, `~` and shift operators `<<`, `>>`
- Operands to these operators can be of any integral type; the type of the result will be the same as the type of the left operand

`&` does bitwise logical **and** of its arguments;

`|` does logical bitwise **or** of its arguments;

`^` does logical bitwise **xor** of its arguments;

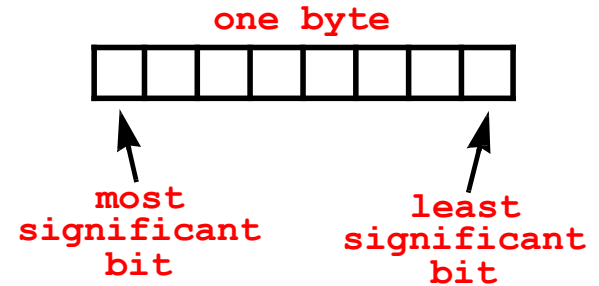
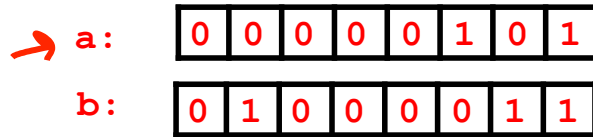
`~` does bitwise logical **complement** of its one argument

`<<` shifts its left argument left by number of bit positions given by its right argument, shifting in 0 on the right;

`>>` shifts its left argument right by number of bit positions given by its right argument, shifting in the sign bit on the left if the left argument is a signed type, else shifts in 0

C++ bitwise operators: examples

unsigned char a = 5, b = 67;



What is the result of $a \& b$

A. 01000111

☒ B. 00000001

C. 01000110

D. Something else

C++ bitwise operators: examples

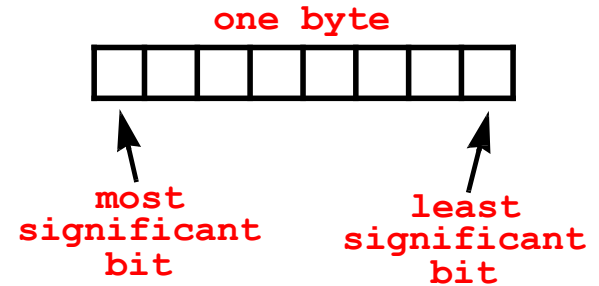
`unsigned char a = 5, b = 67;`

a:

0	0	0	0	0	1	0	1
---	---	---	---	---	---	---	---

b:

0	1	0	0	0	0	1	1
---	---	---	---	---	---	---	---



What is the result of `b >> 5`

- ☒ A. 00000010
- ☐ B. 00000011
- ☐ C. 01100000
- ☐ D. Something else

C++ bitwise operators: examples

unsigned char a = 5, b = 67;

a:

0	0	0	0	0	1	0	1
---	---	---	---	---	---	---	---

b:

0	1	0	0	0	0	1	1
---	---	---	---	---	---	---	---

✓ a & b

0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---

✓ a | b

0	1	0	0	0	1	1	1
---	---	---	---	---	---	---	---

✓ ~a

1	1	1	1	1	0	1	0
---	---	---	---	---	---	---	---

a << 4

0	1	0	1	0	0	0	0
---	---	---	---	---	---	---	---

✓ b >> 1

0	0	1	0	0	0	0	1
---	---	---	---	---	---	---	---

(b >> 1) & 1

0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---

a | (1 << 5)

0	0	1	0	0	1	0	1
---	---	---	---	---	---	---	---

0x1 0 0 0 0 0 0 0 1

b >> 1

one byte

0	0	1	0	0	0	0	1
---	---	---	---	---	---	---	---

most significant bit

least significant bit

5 4 3 2 1 0
0 0 0 0 0 0 1
0 0 1 0 0 0 0

→ getting the second bit in 'b'

C++ bitwise operators: an exercise

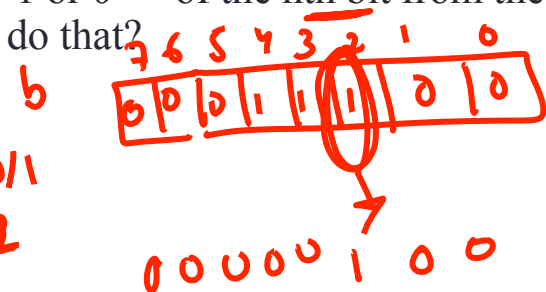
- Selecting a bit: Suppose we want to return the value --- 1 or 0 --- of the nth bit from the right of a byte argument, and return the result. How to do that?

```
byte bitVal(char b, int n) {
```

$c = b \& (1 \ll n)$; $| \& (b \gg n)$ $n=0/1$
 $n=2$

```
    return
```

```
}
```



- Setting a bit: Suppose we want to set the value --- 1 or 0 --- of the nth bit from the right of a byte argument, leaving other bits unchanged, and return the result. How to do that?

```
byte setBit(char b, int bit, int n) {
```

```
    return
```

```
}
```

Defining classes for bitwise I/O

- For a nice object-oriented design, let's define a class **BitOutputStream** that delegates to an object of an existing iostream class, and that adds a **writeBit** method (and a **flush** method which flushes the 8-bit buffer)
- If instead **BitOutputStream** subclassed an existing class, it would inherit all the existing methods of its parent class, and so they become part of the subclass's interface also
 - some of these methods might be useful, but...
 - in general it will complicate the interface
- Otherwise the two design approaches are very similar to implement, except that:
 - with inheritance, **BitOutputStream** uses superclass methods to perform operations
 - with delegation, **BitOutputStream** uses methods of a contained object to perform operations
- We will also consider a **BitInputStream** class, for bitwise input

Outline of a BitOutputStream class using delegation

```

#include <iostream>
class BitOutputStream {
private:
    char buf;                // one byte buffer of bits
    int nbits;                // how many bits have been written to buf
    std::ostream & out;      // reference to the output stream to use
public:

    /** Initialize a BitOutputStream that will use
     * the given ostream for output.
     */
    BitOutputStream(std::ostream & os) : out(os), buf(0), nbits(0) {
        // clear buffer and bit counter
    }

    /** Send the buffer to the output, and clear it */
    void flush()
    {
        out.put(buf);
        out.flush();
        buf = nbits = 0;
    }
}

```

ofstream out;
 out.open(" ");
 BitOutputStream bout(out);

Outline of a BitOutputStream class using delegation, cont

```

/** Write the least significant bit of the argument to
 * the bit buffer, and increment the bit buffer index.
 * But flush the buffer first, if it is full.
 */
void writeBit(int i) {
    // Is the bit buffer full? Then flush it

    // Write the least significant bit of i into the buffer
    // at the current index

    // Increment the index
    nbits++;
}

```

```

char buf✓
int nbits✓
ostream out✓

```

Outline of a BitInputStream class, using delegation

```

#include <iostream>
class BitInputStream {
private:
    char buf;           // one byte buffer of bits
    int nbits;          // how many bits have been read from buf
    std::istream & in    // the input stream to use
public:

    /** Initialize a BitInputStream that will use
     * the given istream for input.
     */
    BitInputStream(std::istream & is) : in(is)
    { buf = 0; // clear buffer
      nbits = ?? // initialize bit index
    }

    /** Fill the buffer from the input */
    void fill() {
        buf = in.get();
        nbits = 0;
    }

```

What should we initialize nbits to?

- ☒ A. 0 ✓
- ☐ B. 1
- ☐ C. 7
- ☒ D. 8 ✓
- ☐ E. Other

preferred →

Outline of a BitInputStream class, using delegation (cont'd)

```
/** Read the next bit from the bit buffer.
 * Fill the buffer from the input stream first if needed.
 * Return 1 if the bit read is 1;
 * return 0 if the bit read is 0.
 *
 */
int readBit() {
    // If all bits in the buffer are read, fill the buffer first

    // Get the bit at the appropriate location in the bit
    // buffer, and return the appropriate int

    // Increment the index

}
```