CSE 100: C++ IO CONTD.
Huffman: Encode & Decode File I/O

1. Build Huffman Tree ← Freq
2. Encode

1. Build Huffman Tree ← Header
2. Decode

Encode (compress)

Decode (uncompress)

Output

Input

Source file

bits

Should be identical
What is a stream?

Streams are essentially sequences of bytes of infinite length that are buffered.
C++ istream

• The `istream` class introduces member functions common to all input streams (that is, streams used for input into your program)

• Some important ones are:

```cpp
istream& operator>>(type & val);
```

- This is the stream extraction operator, overloaded for many primitive types `type`
- Performs an input operation on an istream generally involving some sort of interpretation of the data (like translating a sequence of numerical characters to a value of a given numerical type)
- Returns a reference to the istream, so extractions can be ‘chained’
  ```cpp
  std::cin >>i>> j;
  ```

```cpp
int get();
```

- Perform basic unformatted input. Extracts a single byte from the stream and returns its value (cast to an `int`)
  ```cpp
  int k = cin.get();
  ```

```cpp
istream& read(char* s, streamsize n);
```

- Perform unformatted input on a block of data. Reads a block of data of `n` bytes and stores it in the array pointed to by `s`
  ```cpp
  char buff[40];
  cin.read((buff,30);
  ```
C++ ostream

- The **ostream** class introduces member functions common to all output streams (streams used for output from your program)

- Some important ones are:

  - **`ostream & operator<<(type & val);`**
    - This is the stream insertion operator. It is overloaded for many primitive types `type`. It performs an output operation on an ostream generally involving some formatting of the data (like for example writing a numerical value as a sequence of characters). It returns a reference to the ostream, so insertions can be ‘chained’.
      
      ```cpp
      std::cout << a << " and " << b << std::endl;
      ```

  - **`ostream & put(char c);`**
    - Perform basic unformatted output. Writes a single byte to the stream and returns a reference to the stream

  - **`ostream & write(const char* s, streamsize n);`**
    - Perform unformatted output on a block of data. Write a block of data of `n` bytes starting at address `s`

  - **`ostream & flush();`**
    - Any unwritten characters in the ostream’s buffer are written to its output destination as soon as possible ("flushed").

  **cout and cerr are instances of ostream**
C++ ifstream and ofstream

- The *ifstream* class inherits from istream, and introduces functions specialized for doing input from files:

  ```cpp
  void open ( const char * filename,
              ios_base::openmode mode = ios_base::in );
  • Opens a file whose name is `filename`.
  ```

  ```cpp
  void close ( );
  • Closes the file associated with the stream. The stream is flushed first
  ```

- The *ofstream* class inherits from ostream and introduces functions specialized for doing output to files:

  ```cpp
  void open ( const char * filename,
              ios_base::openmode mode = ios_base::out );
  • Opens a file whose name is `filename`.
  ```

  ```cpp
  void close ( );
  • Closes the file associated with the stream.
#include <iostream>
#include <fstream>

using namespace std;

int main( int argc, char** argv )
{
    ifstream theFile;
    string nextWord;
    theFile.open( "testerFile.txt" );
    while ( 1 ) {
        theFile >> nextWord;
        if (theFile.eof()) break;  // Also if (!theFile.good()) break
        cout << nextWord << " ";
    }
    theFile.close();
}
#include <iostream>
#include <fstream>

using namespace std;

int main( int argc, char** argv )
{
    ifstream theFile;
    string nextWord;
    theFile.open( "testerFile.txt" );
    while ( 1 ) {
        theFile >> nextWord;
        if (theFile.eof()) break; // Also if (!theFile.good()) break
        cout << nextWord << " ";
    }
    theFile.close();
}

Notice that this code will strip formatting and read whole strings! (Not what you should do for your internal checkpoint…)

Identify the C++ operator that is allowing us to read from the file!
Reading bytes from a file

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

using namespace std;

int main( int argc, char** argv )
{
    ifstream theFile;
    char nextChar;
    theFile.open( "testerFile.txt" );
    while ( 1 ) {
        nextChar = ___________________________;  // What should go in the blank so that we read a character at a time from a text file?
        if (theFile.eof()) break;
        cout << nextChar;
    }
    theFile.close();
}
```

What is the difference between this approach and using the >> operator?

What should go in the blank so that we read a character at a time from a text file?
Writing to a file

- In your Huffman code program you will write the encoded text from the infile to an outfile by writing out the code (a sequence of 0s and 1s) for each character in sequence.
- What is wrong with using with the following method for writing these codes to the file?

```
// assume that outStream is an ofstream, n is an HCNode
// and HCNode has a boolean field isZeroChild
...
if (n->isZeroChild) {
    outStream << '0';
} else {
    outStream << '1';
}
```

A. Nothing
B. You cannot use the `<<` operator to write to a file in C++
C. The ‘compressed’ file will be larger than the uncompressed file
D. The bits in the code will be written in the wrong order
• In your Huffman code program you will write the encoded text from the infile to an outfile by writing out the code (a sequence of 0s and 1s) for each character in sequence.

• What is wrong with using with the following method for writing these codes to the file?

```cpp
// assume that outStream is an ofstream, n is an HCNode
// and HCNode has a boolean field isZeroChild
...
if (n->isZeroChild) {
    outStream << '0';
} else {
    outStream << '1';
}
```

A. Nothing  
B. You cannot use the << operator to write to a file in C++  
C. The ‘compressed’ file will be larger than the uncompressed file  
D. The bits in the code will be written in the wrong order

But this is exactly what you will do for the internal Checkpoint deadline! (We’ll talk about how to write one bit at a time later)
Reading and writing numbers

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

using namespace std;

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

What is in `number.txt` after this program is executed?

Is the binary data written into the file same as that representing the number 12345?

No, in the above example the ascii values of '1', '2', '3', '4', '5' are written to file.
## Writing raw numbers

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

using namespace std;

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

This is the method you’ll use for the final submission (and the checkpoint too if you want)

Recap: Different ways of writing to file
# include <iostream>
#include <fstream>

using namespace std;

int main( int argc, char** argv )
{
    ofstream numFile;
    int num = 12345;
    numFile.open( "number.txt" );  // .txt isn't really correct anymore...
    numFile.write( (char*)&n, sizeof(n) ) ;
    numFile.close();
    // Getting the number back!
    ifstream numFileIn;
    numFileIn.open( "number.txt" );
    int readN;
    numFileIn.read((char*)&readN, sizeof(readN));
    cout << readN << endl;
    numFileIn.close();
}
# Opening a binary file

```cpp
#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
• 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.
First write/read disk is slow
we want to be able to write
b\text{its}
BitOutputStream:

encoder → Buffer → ostream

BitInputStream:

istream → Buffer → decoder

DATA IN

4KB

DATA OUT
Buffers, Latency and Throughput

- **Latency**: the time between a new input and the corresponding output.
  - Lower is better.
- **Throughput**: the number of input bytes processed per second.
  - Higher is better.
- Buffers hurt (increase) latency but improve (increase) throughput.
  - Consider traveling SD->SF using a train vs. a private car.
  - Consider express lane vs. regular lane in the super-market.
Why Buffer?

Q: Why should we implement buffering in our BitOutputStream and BitInputStream class for the Huffman coding problem?

A. To improve latency
B. To improve throughput
C. To use the standard file IO stream interfaces provided by C++
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)
• C++ I/O
• I/O buffering
• Bit-by-bit I/O
C++ bitwise operators

- C++ has bitwise logical operators \&, |, ^, ~ and shift operators \ll, \gg

- 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

- \ll shifts its left argument left by number of bit positions given by its right argument, shifting in 0 on the right;
- \gg 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;`

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a &amp; b</code></td>
<td>0 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>`a</td>
<td>b`</td>
</tr>
<tr>
<td><code>~a</code></td>
<td>1 1 1 1 1 0 1 0</td>
</tr>
<tr>
<td><code>a &lt;&lt; 4</code></td>
<td>0 1 0 1 0 0 0 0</td>
</tr>
<tr>
<td><code>a &gt;&gt; 1</code></td>
<td>0 0 1 0 0 0 0 1</td>
</tr>
<tr>
<td><code>(b &gt;&gt; 1) &amp; 1</code></td>
<td>0 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>`a</td>
<td>(1 &lt;&lt; 5)`</td>
</tr>
</tbody>
</table>
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?

```c++
byte bitVal(char b, int n) {
    return (b >> n) & 0x1;
}
```

• 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?

```c++
byte setBit(char b, int bit, int n) {
    b = b & ~((1 << n)); // clears bit at position n
    return b | (bit << n); // sets nth bit to the value 'bit'
}
```
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

```cpp
#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;
    }
};
```
Outline of a BitOutputStream class, using delegation (cont’d)

/** 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

}
#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 = 0;  // initialize bit index
    }

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

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
A very bad way is to start at the root and search down the tree until you find the symbol you are trying to encode.
Encoding a symbol

A much better way is to maintain a list of leaves and then to traverse the tree to the root (and then reverse the code… or not?).

```cpp
vector<HCNode*> leaves;
...
leaves = vector<HCNode*>(256, (HCNode*)0);
```
class LNode {
    int data;
    LNode* next;
}

Assume you have created the following list:

```cpp
void traverse(LNode* n) {
    while(n) {
        std::cout << n->data << std::endl;
        n = n->next;
    }
}
```

What does `traverse(first)` print?

A. 1 2 3
B. 3 2 1
C. Something else
void traverse(LNode* n) {
    // 1
    if (n == 0) return;
    // 2
    traverse(n->next);
    // 3
}

Where should I put the line to print \texttt{n->data} to print the list in reverse order?
\begin{verbatim}
std::cout << n->data << std::endl;
\end{verbatim}
A. 1  B. 2  C. 3
A much better way is to maintain a list of leaves and then to traverse the tree to the root (and then reverse the code... or not?).

Use recursion to easily write a symbol’s code in the correct order!

vector<HCNode*> leaves;
...
leaves = vector<HCNode*>(256, (HCNode*)0);