

CSE 100:
STREAM I/O, BITWISE

OPERATIONS,
BIT STREAM I/O

PA2: encoding/decoding

ENCODING:

1. Scan text file to compute frequencies

→ 2. **Build Huffman Tree**

3. Find code for every symbol (letter)

4. Create new compressed file by saving the entire code at the top of the file followed by the code for each symbol (letter) in the file

DECODING:

1. Read the file header (which contains the code) to recreate the tree

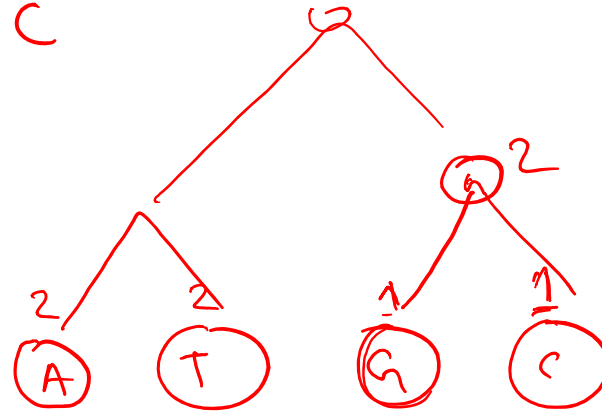
2. Decode each letter by reading the file and using the tree

Today's Class

- Build Huffman Tree
- C++ I/O
- I/O buffering
- Bit-by-bit I/O

Building Huffman Tree

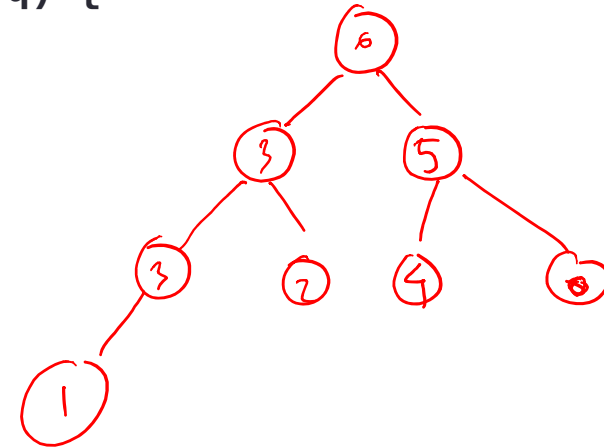
G A A T T C



~~A~~	2
T	2
G	1
C	1

Priority Queues in C++

```
template<typename T> void print_queue(T& q) {
    while(!q.empty()) {
        std::cout << q.top() << " ";
        q.pop();
    }
    std::cout << '\n';
}
```



```
int main() {
    std::priority_queue<int> q;
    for(int n : {1,8,5,6,3,4,0,9,3,2})
        q.push(n);
    print_queue(q);
```

9 8 6 5 4 3 3 2 1 0

```
std::priority_queue<int, std::vector<int>, std::greater<int> > q2;
```

```
for(int n : {1,8,5,6,3,4,0,9,3,2})
    q2.push(n);
    print_queue(q2);
```

0 1

```
}
```

Using `std::priority_queue` in Huffman's algorithm

- If you create an STL container such as `priority_queue` to hold `HCNode` objects:

```
#include <queue>
```

```
std::priority_queue<HCNode> pq;
```

- ... then adding an `HCNode` object to the `priority_queue`:

```
HCNode n;
```

```
pq.push(n);
```

- ... actually creates a copy of the `HCNode`, and adds the copy to the queue.

In HCNODE.cpp:

```
#include HCNODE_HPP
/** Compare this HCNODE and other for priority
ordering.
 * Smaller count means higher priority.
 * Use node symbol for deterministic tiebreaking
 */
bool HCNODE::operator<(HCNODE const & other) const {
    // if counts are different, just compare counts
    if(count != other.count) return count >
other.count;
    // counts are equal. use symbol value to break tie.
    // (for this to work, internal HCNODEs
    // must have symb set.)
    return symb < other.symb;
};

#endif
```

Using `std::priority_queue` in Huffman's algorithm

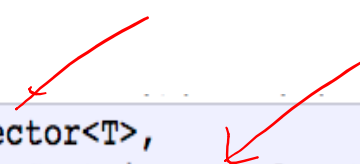
```
std::priority_queue<HCNode*↙> pq;  
HCNode* p = new HCNode();  
pq.push(p);
```

l h
1 l < h)
c1

std::priority_queue template arguments

- The template for priority_queue takes 3 arguments:
- The template for priority_queue takes 3 arguments:

```
1 template < class T, class Container = vector<T>,
2           class Compare = less<typename Container::value_type> > class priority_queue;
```



- The first is the type of the elements contained in the queue.
- If it is the only template argument used, the remaining 2 get their default values:
 - a **vector<T>** is used as the internal store for the queue,
 - **less** a class that provides priority comparisons
- Okay to use vector container , but we want to tell the priority_queue to first dereference the HCNode pointers it contains, and then apply operator<
- How to do that? We need to provide the priority_queue with a Compare class

Defining a “comparison class”

- The documentation says of the third template argument:
- Compare: Comparison class: A class such that the expression `comp(a,b)`, where `comp` is an object of this class and `a` and `b` are elements of the container, returns true if `a` is to be placed earlier than `b` in a strict weak ordering operation. This can be a class implementing a function call operator...

Here's how to define a class implementing the function call operator `operator()` that performs the required comparison:

```
class HCNODEPtrComp {  
    bool operator()(HCNode* & lhs, HCNODE* & rhs) const {  
        // dereference the pointers and use operator<  
        return *lhs < *rhs;  
    }  
};
```

Already done in the starter code

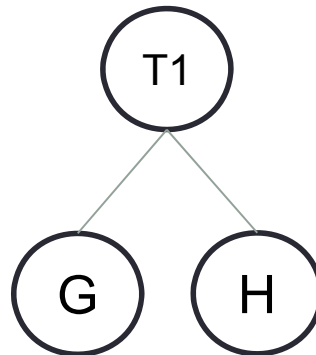
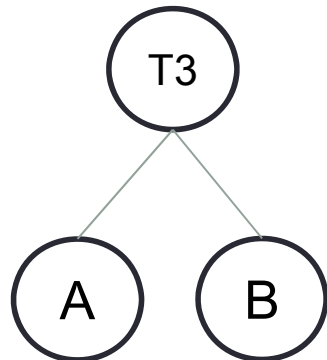
Now, create the `priority_queue` as:

```
std::priority_queue<HCNode*, std::vector<HCNode*>, HCNODEPtrComp> pq;
```

and priority comparisons will be done as appropriate.

One more piece

- When you create a new node in the forest, what character should it hold (beyond holding the sum of their children)?
 - A. The character '0'
 - ☒ B. The alphabetically smaller character of its children
 - ☒ C. The alphabetically larger character of its children
 - D. It doesn't matter



PA2: encoding/decoding

These require file I/O.
We'll talk about that today.

ENCODING:

- 1. Scan text file to compute frequencies
2. Build Huffman Tree
3. Find code for every symbol (letter)
- 4. Create new compressed file by saving the entire code at the top of the file followed by the code for each symbol (letter) in the file

DECODING:

1. Read the file header (which contains the code) to recreate the tree
2. Decode each letter by reading the file and using the tree

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 (retain the ability to output in ASCII, it may come in handy)
- Compress/decompress a small file (100 bytes)
- Decompression should map the encoded input back to the original input

C++ istream

std::cin

- The `istream` class introduces member functions common to all input streams (that is, streams used for input into your program)
- Some important ones are:

`cin` is an instance of `istream`

`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'

`std::cin >>i>>j;`

`int get();`

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

`int k = cin.get( );`

`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`

`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:

cout and **cerr** are instances of **ostream**

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'.

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").

C++ ifstream and ofstream

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

```
void open ( const char * filename,
            ios_base::openmode mode = ios_base::in );
```

- Opens a file whose name is **filename**.

```
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:

```
void open ( const char * filename,
            ios_base::openmode mode = ios_base::out );
```

- Opens a file whose name is **filename**.

```
void close ( );
```

- Closes the file associated with the stream.

Reading *bytes* from a file

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

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

```
using namespace std;
```

```
int main( int argc, char** argv )
{
```

```
    ifstream theFile;
```

```
    char nextChar;
```

```
    theFile.open( "testerFile.txt" );
```

```
    while ( 1 ) {
```

```
        nextChar = _____;
```

```
        if (theFile.eof()) break;
```

```
        cout << nextChar;
```

```
    }
```

```
    theFile.close();
```

```
}
```

☒ A. theFile.get();

☒ B. (char)theFile.get();

☒ C. (int)theFile.get();

☐ D. theFile.get(1);

☒ E. More than one will work

Writing bytes 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 for actual compression?

```
// 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';
}
```

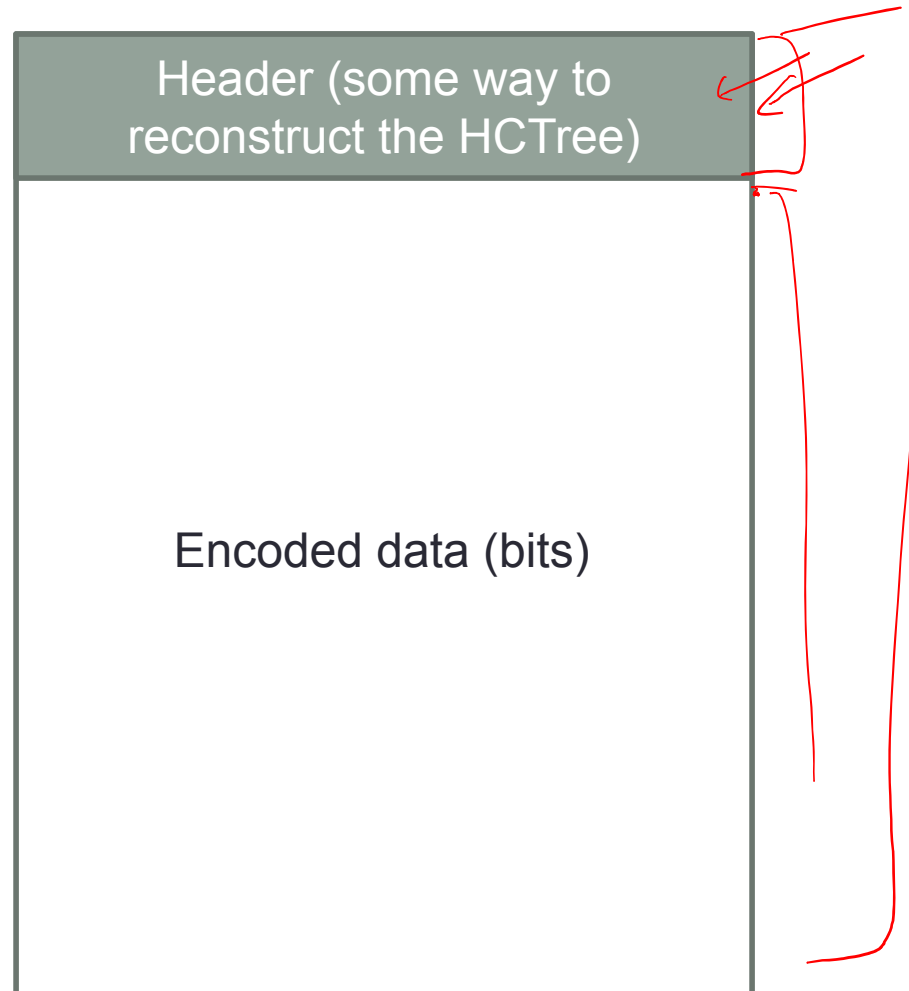
- A. Nothing
- B. You cannot use the << operator to write to a file in C++
- C. The 'compressed' file will be larger than the uncompressed file
- D. The bits in the code will be written in the wrong order

Writing the compressed file

In the reference solution, the header is just 256 ints in a row, one for each byte value.

This header takes 1024 bytes of space. (4×256)

You must beat this for your final submission.



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();
}
```

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();
}
```

45

12345.45

You'll need to include delimiters between the numbers

↙

Writing raw numbers

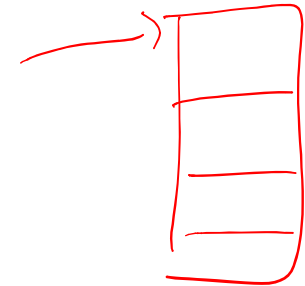
```
#include <iostream>
#include <fstream>

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();
}
```

Handwritten notes:

- 4 bytes (written above the `sizeof(num)` argument)
- 4 (written below the `sizeof(num)` argument)



This is the method you'll use for the final submission
Let's look at the file after we run this code...

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;
    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();
}
```

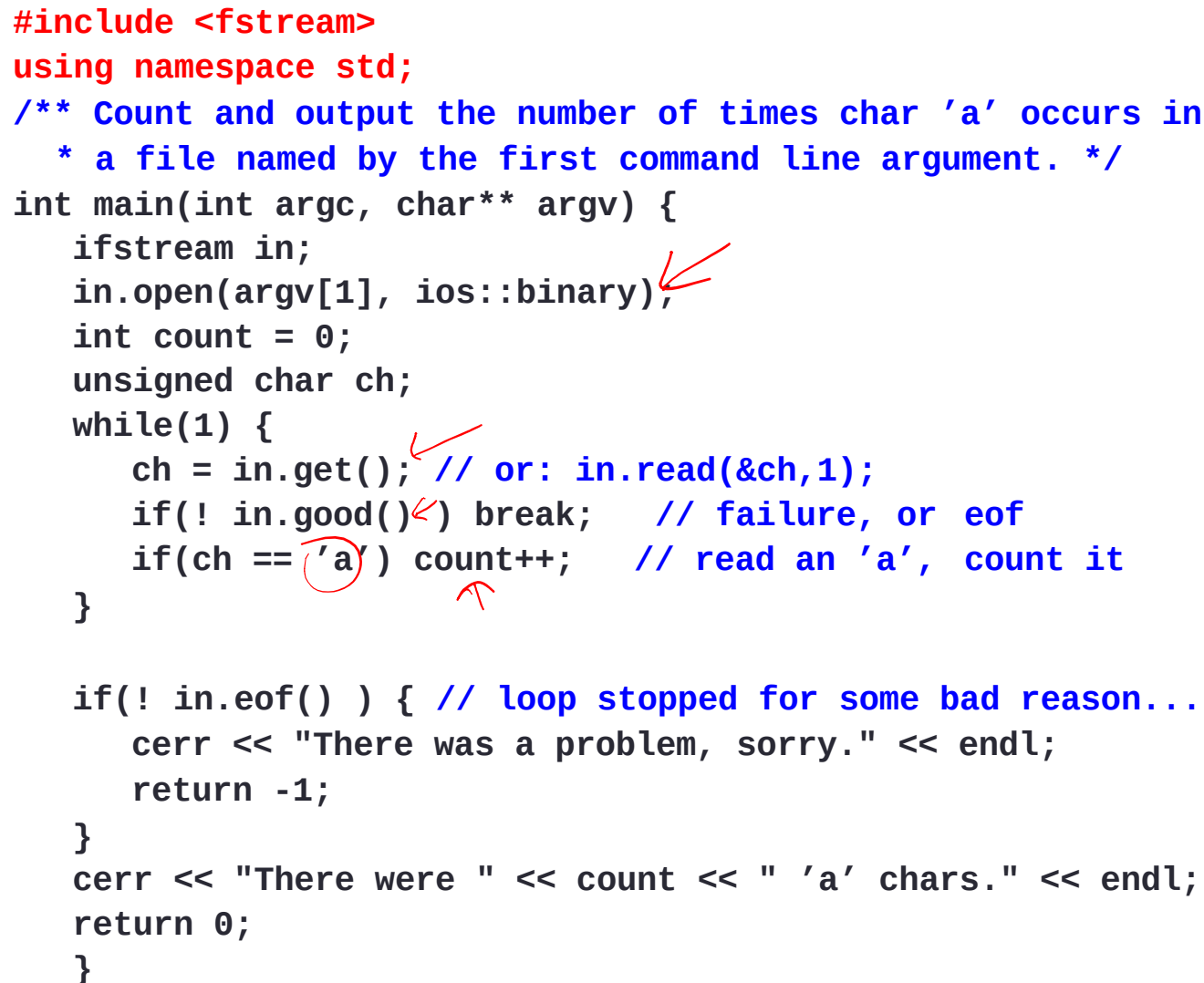
Binary and nonbinary file streams

- 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;
}
```



Writing the compressed file

Now let's talk about how to write the bits...

