CSE 100: I/O IN C++

Goals for Today

- Huffman Tree
- Use C++ I/O classes
- Trace the process of Huffman coding

PA3: 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

PA3: encoding/decoding

ENCODING:

1.Scan text file to compute frequencies

2.Build Huffman Tree

3.Find code for every symbol (letter)

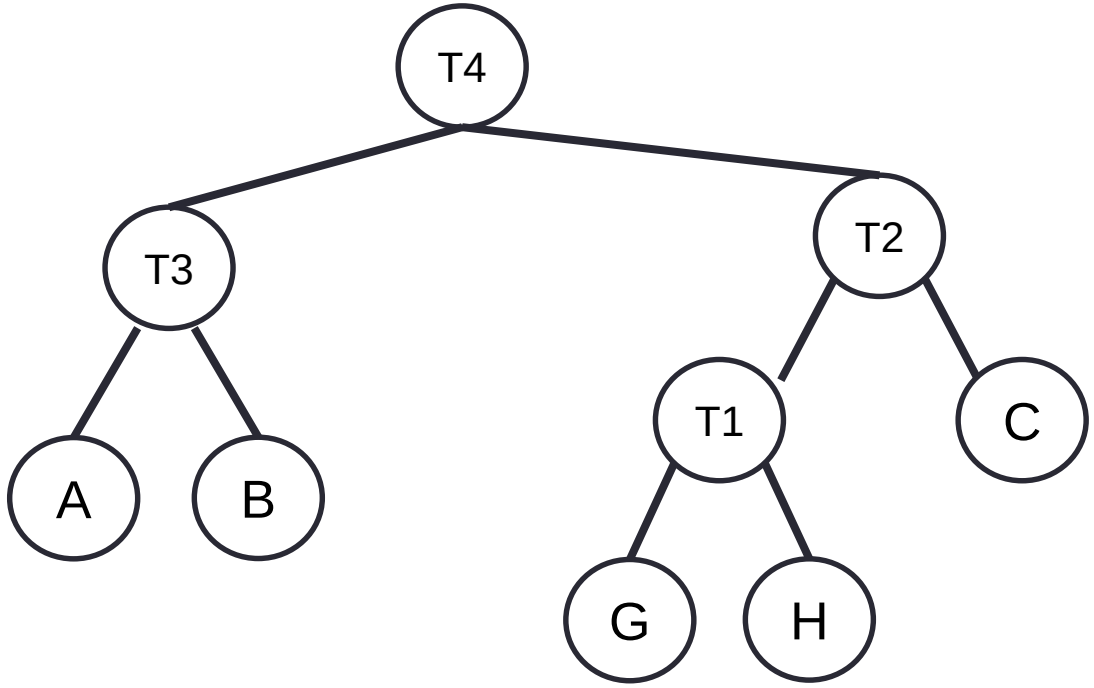
4.Create new compressed file by saving the entire code at the top of the file followed by the code for each symbol (letter) in the file

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

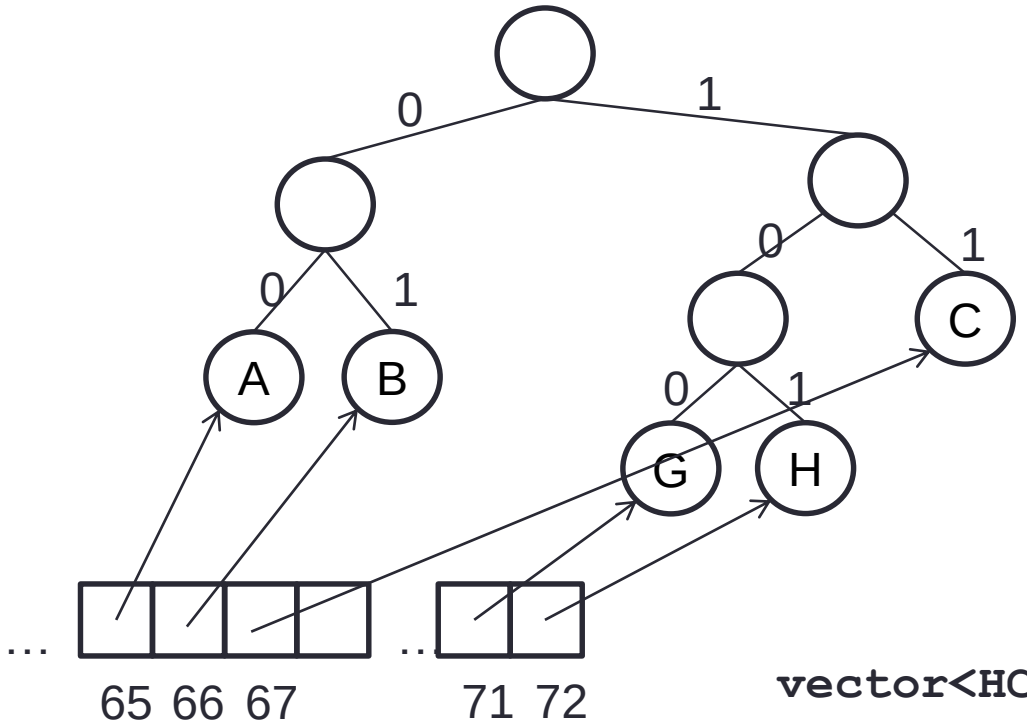
Encoding a symbol: let's think implementation!



- Compression using trees:
  - Devise a “good” code/tree
  - Encode symbols using this tree

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, why?

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)

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

PA3: 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

Building the tree: Huffman's algorithm

0. Determine the count of each symbol in the input message.
1. Create a forest of single-node trees containing symbols and counts for each non-zero-count symbol.
2. Loop while there is more than 1 tree in the forest:

   2a. Remove the two lowest count trees

   2b. Combine these two trees into a new tree (summing their counts). 2c. Insert this new tree in the forest, and go to 2.
3. Return the one tree in the forest as the Huffman code tree.

You know how to create a tree. But how do you maintain the forest? Choose the best data structure/ADT:

A. A list
B. A BST
C. A priority queue (heap)

Aside: Heaps

Have you seen a heap? A. Yes B. No C. Yes, but I don't remember them

Priority Queues in C++

A C++ **`priority_queue`** is a generic container, and can hold any kind of thing as specified with a template parameter when it is created: for example **`HCNode`**s, or pointers to **`HCNode`**s, etc.

```
#include <queue>
std::priority_queue<HCNode> p;
```

By default, a **`priority_queue<T>`** uses **`operator<`** defined for objects of type T:

- **if `a < b`, `b` is taken to have higher priority than `a` and `b` will come out before `a`**

Priority Queues in C++

```
#ifndef HCNODE_H
#define HCNODE_H
class HCNode {

public:
  HCNode* parent; // pointer to parent; null if root
  HCNode* child0; // pointer to "0" child; null if leaf
  HCNode* child1; // pointer to "1" child; null if leaf
  unsigned char symb; // symbol
  int count; // count/frequency of symbols in subtree

  // for less-than comparisons between HCNodes
  bool operator<(HCNode const &) const;
};

#endif
```

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
```

Is this implementation of operator< correct to use with the C++ priority queue (which uses a MAX-heap)?

A. Yes
B. No

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
```

If you have two HCNode* pointers in the priority queue, one with symbol ‘A’ and the other with symbol ‘D’, both with frequency (count) 200, which will come out first?

A. The one with symbol ‘A’
B. The one with symbol ‘D’
C. You can’t tell, it could be either

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. You probably don't want that. Instead, set up the container to hold pointers to HCNode objects:

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

Using std::priority_queue in Huffman's

Instead, set up the container to hold pointers to HCNode objects:

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

What is the problem with the above approach?

A. Since the priority queue is storing copies of HCNode objects, we have a memory leak

B. The nodes in the priority queue cannot be correctly compared

C. Adds a copy of the pointer to the node into the priority queue

D. The node is created on the run time stack rather than the heap

Using std::priority_queue in Huffman's algorithm

Instead, set up the container to hold pointers to HCNode objects:

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

What is the problem with the above approach?

- our operator< is a member function of the HCNode class. It is not defined for pointers to HCNodes. What to do?

std::priority_queue template 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() 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.

PA3: encoding/decoding

ENCODING:

1.Scan text file to compute frequencies (Monday will help)

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 (next lecture will help)

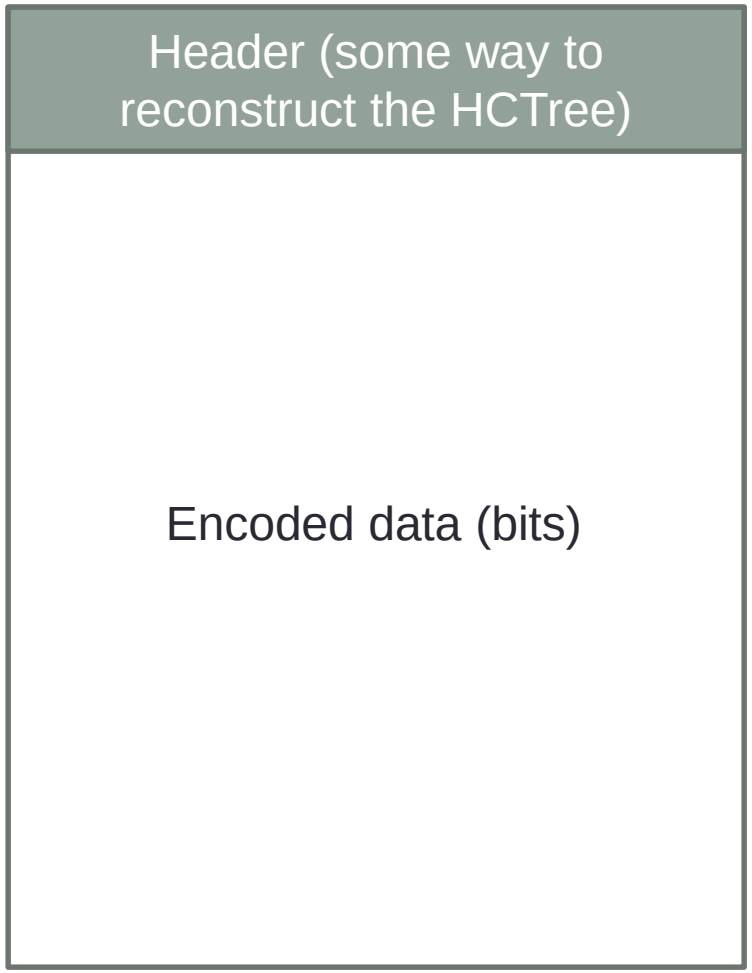
The File Header: A way to recreate the tree

Writing the compressed file

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

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

You must beat this for your final submission.



C++ istream

cin is an instance of *istream*

You will use the istream to get data from a file (actually ifstream, which inherits from istream)
But which method(s)/operators to use for reading??

istream& operator>> (type & val); Formatted data (e.g. reads text-based numbers as numbers)

int get(); Basic unformatted data (single character)

istream& read (char s, streamsize n);* More general unformatted data (can be more than a byte)

C++ ostream

cout and *cerr* are instances of *ostream*

You will use the ostream to write data to a file (actually ofstream, which inherits from ostream)
But which method(s)/operators to use for writing??

ostream& operator<< (type & val); Formatted data (e.g. outputs numbers as text)

ostream& put(char c); Basic unformatted output (single character)

ostream& write(const char c, streamsize n);* More general unformatted output (any size)

ostream& flush(); Write out any unwritten bits

Reading raw *bytes* from a file

Does this work to read all of the bytes from a file correctly?

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

using namespace std;

int main( int argc, char** argv )
{
  ifstream theFile;
  unsigned char nextChar;
  theFile.open( "testerFile.txt" );
  while (!theFile.eof()) {
    nextChar = theFile.get();
    cout << nextChar << "#";

  }
  cout << endl;
  theFile.close();
}
```

A. Yes
B. No

Reading raw *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;
  unsigned char nextChar;
  theFile.open( "testerFile.txt" );
  while ( 1 ) {
    nextChar = _____________________________;
    if (theFile.eof()) break;
    cout << nextChar << "#";
  }
  cout << endl;
  theFile.close();
}
```

A. `theFile.get();`
B. `(unsigned char)theFile.get();`
C. `(int)theFile.get();`
D. `theFile.get(1);`
E. More than one will work

Reading from a file, alternate approach

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

using namespace std;

int main( int argc, char** argv )
{
  ifstream theFile;
  unsigned char nextChar;
  int nextByte;
  theFile.open("testerFile.txt", ios::binary);
  while ((nextByte = theFile.get()) != EOF) {
    nextChar = (unsigned char)nextByte;
    cout << nextByte << endl;
    cout << nextChar << endl;
  }
  theFile.close();
}
```

Why use an int??

Formatted output: 1's and 0's for the checkpoint

```
// 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';
}
```

Creates a BIGGER "compressed" file

Reading and writing numbers

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

using namespace std;

int main()
{
  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? (What are the actual bits in the file?)

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

du –b numfile
hexdump –C numfile

What is the raw data in this file?

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

using namespace std;

int main()
{
  ofstream numFile;
  int num = 12345;
  numFile.open( "numfile" );
  numFile << num;
  numFile.close();
}
```

Dec	Oct	Hex	Bin	Symb
49	061	31	0011 0001	1
50	062	32	0011 0010	2
51	063	33	0011 0011	3
52	064	34	0011 0100	4
53	065	35	0011 0101	5

hexdump –C numfile

Write out the file content (naive solution)

The file to be encoded include
AABCBA

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();
}
```

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

cat numfile
hexdump –b numfile

Reading raw numbers

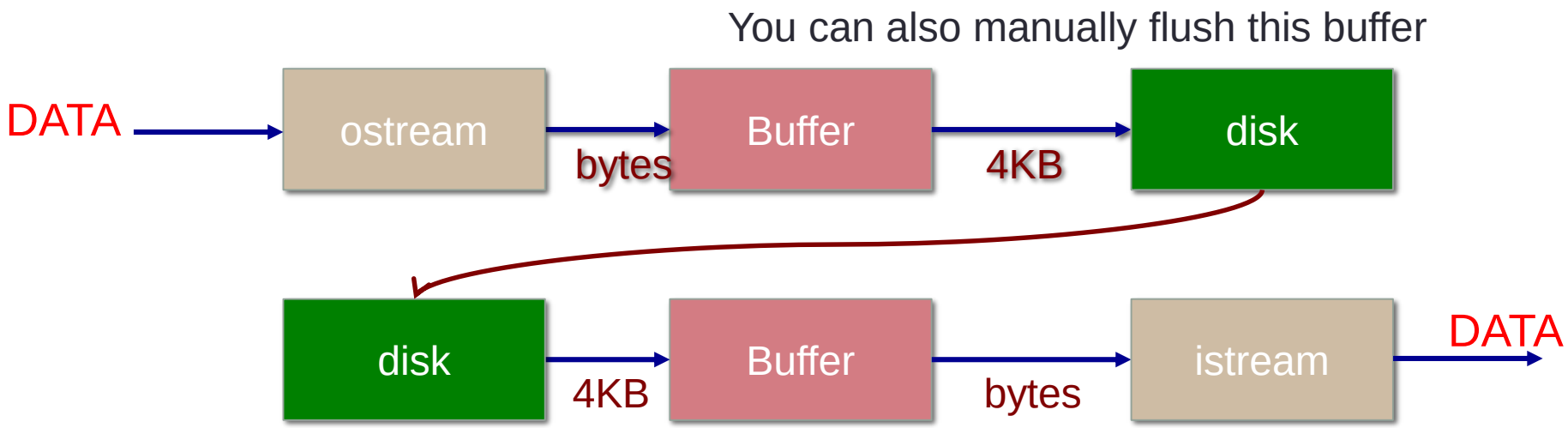
```
#include <iostream>
#include <fstream>

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();
}
```

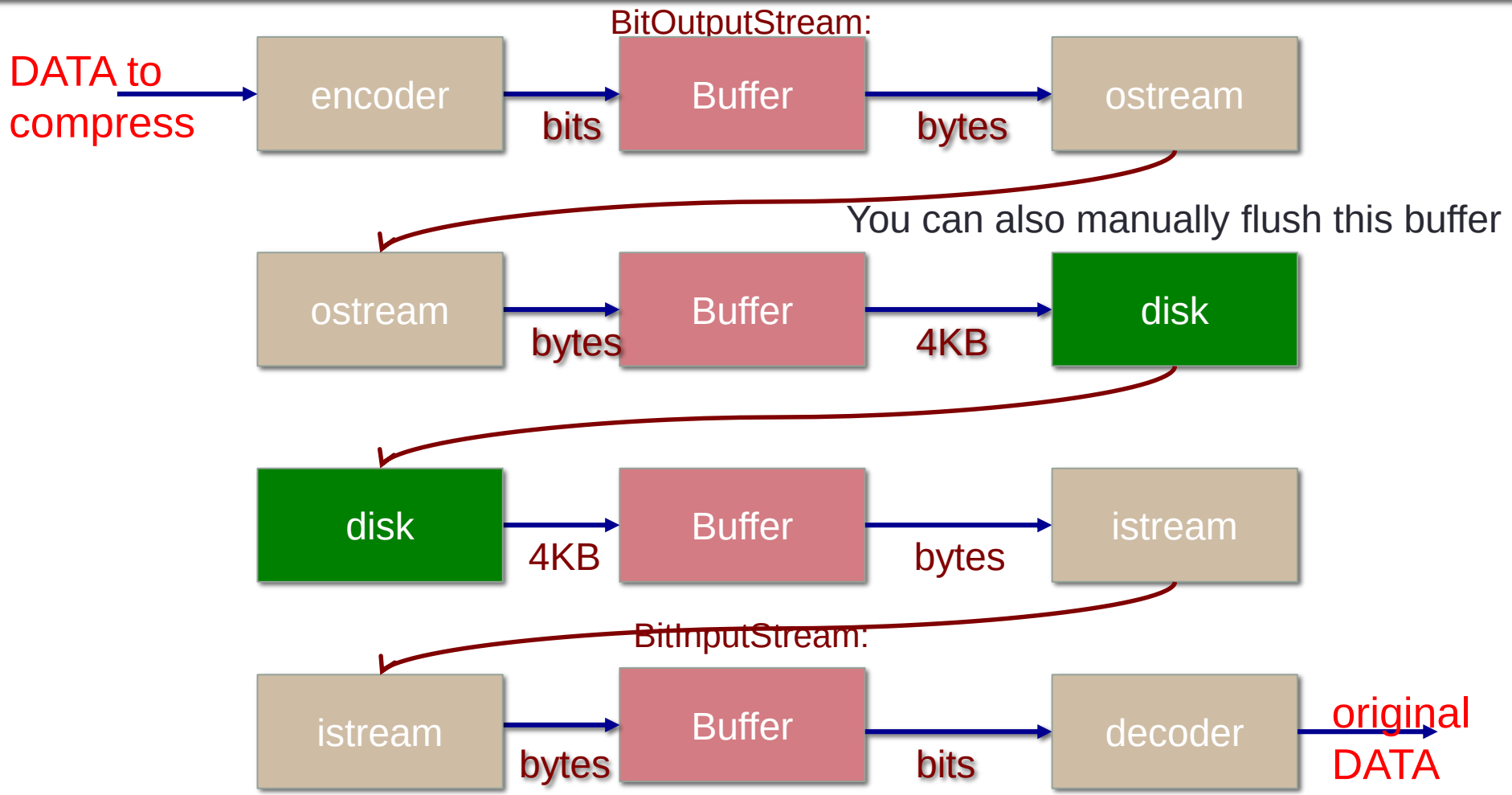
Streams and Buffers



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)

Streams and Buffers



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;

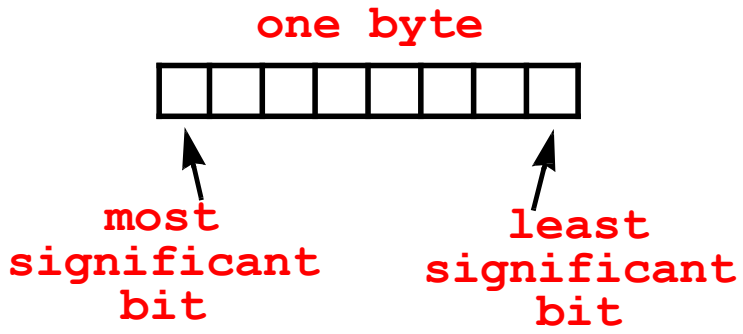
>> shifts its left argument right by number of bit positions given by its right argument, shifting in the sign bit on the left if the left argument is a signed type, else shifts in 0

C++ bitwise operators: examples

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

0	0	0	0	0	1	0	1

0	1	0	0	0	0	1	1



What is the result of `a & b`

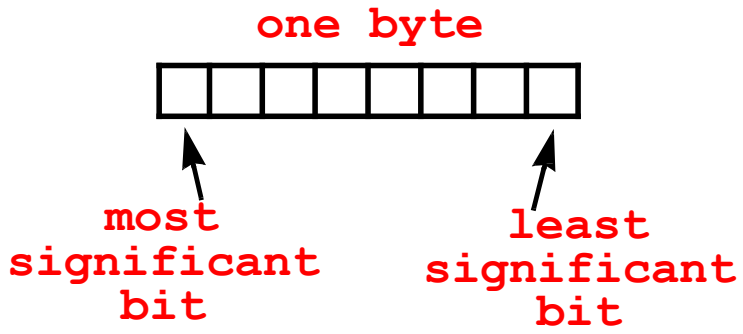
A. 01000111
B. 00000001
C. 01000110
D. Something else

C++ bitwise operators: examples

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

0	0	0	0	0	1	0	1

0	1	0	0	0	0	1	1



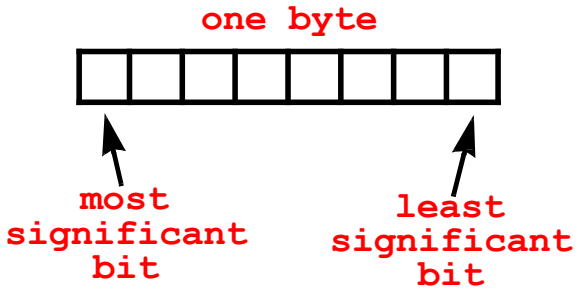
What is the result of `b >> 5`

A. 00000010
B. 00000011
C. 01100000
D. Something else

C++ bitwise operators: examples

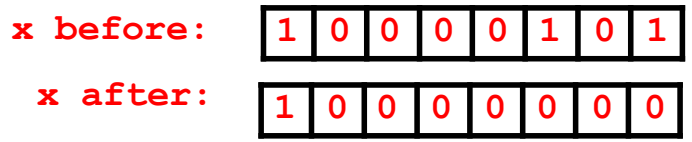
```
unsigned char a  = 5, b = 67;
```

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

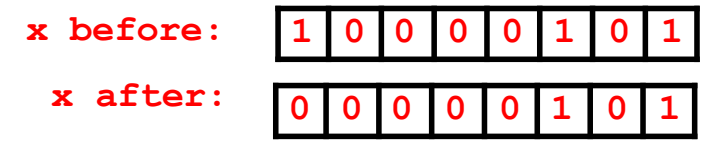


C++ bitwise operators: practice

Write a statement that will zero-out every bit *except* for the leftmost bit of some byte x.



Write a statement that will zero-out *only* the leftmost bit of some byte x.



&	bitwise and
|	bitwise or
^	bitwise xor
~	bitwise complement
>>	shift right
<<	shift left

C++ bitwise operators: an exercise

Could be char; doesn't matter

- 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(byte b, int n) {



}
```

- 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(byte b, int bit, int n) {



}
```

typedef unsigned char byte;

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;
    }
```

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 BitOutputStream::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

}
```

```
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

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  appriopriate location  in the bit
   // buffer, and  return  the appropriate int



   // Increment the index

}
```

```
char buf
int nbits
istream& in
```