Lecture 10

• C++ I/O

• Some useful classes in `<iostream>`

• I/O buffering

• Bit-by-bit I/O

Reading: online documentation on C++ streams
A quick tour of the C++ I/O classes

• The C++ standard library defines classes used for I/O

• We will look at some of the most important classes in that package, to understand how and why to use them
  • (For more details, see the online documentation)

• All C++ I/O class names are in the std namespace

• All C++ I/O classes inherit from the class std: :ios_base

• We will mainly concentrate on the classes used for file I/O...
Class inheritance hierarchy for the C++ I/O classes

- (At the top of each column is shown the system header file that should be included in order to use the classes in that column)
- Note that \texttt{cout}, \texttt{cerr} are instances of \texttt{ostream}; and \texttt{cin} is an instance of \texttt{istream}
- In fact, the \texttt{\textless\textless} operator for output, and the \texttt{\textgreater\textgreater} operator for input, work for any \texttt{ostream} or \texttt{istream} object, respectively
- However, these operators do “formatted” I/O. What if you want to do raw binary I/O?
**C++ iostream object error conditions**

- When dealing with streams, various errors and exceptional conditions can happen (good things happen too, hopefully)
  - The stream might be corrupted or incapable of working, in some unrecoverable way
  - A particular operation might have failed, but the stream is recoverable
  - On input, the end of stream (EOF) might be reached

- With these C++ classes, these conditions are signaled by *setting flags on the stream object*, which can be inspected with member functions

- These functions are defined in the `ios` class, so are inherited by all I/O stream objects:
  ```cpp
define bad ( ) const; // return true if "bad" flag is set
bool fail ( ) const; // return true "bad" or "fail" flag is set
bool eof ( ) const; // return true if "eof" flag is set
bool good ( ) const; // return true if no flag is set
void clear ( ); // clear (i.e. unset) all flags
```

- (It is also possible to have the stream object throw an exception instead of setting a flag; see documentation for details)
C++ `istream`

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

```
istream& operator>>(type & val);
```
  - This is the stream extraction operator. It is overloaded for many primitive types `type`. It 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). It returns a reference to the istream, so extractions can be ‘chained’.

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

```
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`. 

C++ ostream

• The ostream class introduces member functions in common to all output streams (that is, 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’.

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 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 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.
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
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;
    char ch;
    while(1) {
        ch = in.get(); // or: in.read(&ch,1);
        if(! in.good() ) break; // failure, or eof
        if(ch == 'a') count++;
    }
    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;
#include <fstream>
using namespace std;
/** Given a file containing whitespace delimited decimal integers
   * named by the first command line argument, output their sum */
int main(int argc, char** argv) {
    ifstream in;
in.open(argv[1]);
    int sum = 0, n;
    while(1) {
        in >> n; // read the next integer from the ifstream
        if(! in.good() ) break; // failure, or eof
        sum += n; // accumulate it in the sum
    }
    if(! in.eof() ) { // loop stopped for some bad reason...
        cerr << "There was a problem, sorry." << endl; return -1;
    }
    cerr << "The sum is: " << sum << endl;
    return 0;
Formatted vs unformatted file output: an example

```cpp
#include <fstream>
#include <iomanip>

int main() {
    double d = 3.1415926535;
    ofstream of1;
    of1.open("out1", ios::binary); // to force 1 byte per char
    of1 << setw(12) << d;
    of1.close();

    ofstream of2;
    of2.open("out2", ios::binary);
    of2.write((char*)&d, sizeof(d));
    of2.close();

    • What is the resulting size of file `out1`? _____ bytes
      • What does it contain?

    • What is the resulting size of file `out2`? _____ bytes
      • What does it contain?
```
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
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 that adds writeBit or readBit methods (and a flush method which flushes the 8-bit buffer)
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;

a: \begin{array}{c}
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
1 \\
1 \\
\end{array}

b: \begin{array}{c}
0 \\
1 \\
0 \\
0 \\
0 \\
0 \\
0 \\
1 \\
\end{array}

\begin{align*}
a \& b &= 00000000011 \\
da | b &= 010001111 \\
\sim a &= 111110100 \\
a \ll 4 &= 010100000 \\
b \gg 1 &= 00100001 \\
(b \gg 1) \& 1 &= 000000001 \\
a | (1 \ll 5) &= 001001011
\end{align*}
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
  }
  ```

- 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) {
    return
  }
  ```
Defining classes for bitwise I/O

• For a nice object-oriented design, let’s define a class \texttt{BitOutputStream} that delegates to an object of an existing iostream class, and that adds a \texttt{writeBit} method (and a \texttt{flush} method which flushes the 8-bit buffer)

• If instead \texttt{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 \texttt{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 = nbits = 0;  // clear buffer and bit counter
        }

        /** Send the buffer to the output, and clear it */
        void flush() {
            os.put(buf);
            os.flush();
            bit_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
}
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.
    */
    BitOutputStream(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;
    }
}
/** 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

}
The std::bitset class template

- The STL provides the `bitset` class template that can be useful when needing to manipulate individual bits
  - `#include <bitset>` to access the relevant declarations

- The `bitset` class template takes one template parameter: an integer, specifying how many bits the bitset contains. So to create a bitset containing 8 bits:

  ```
  bitset<8> buf;
  ```

- By default, a bitset is created with all its bits 0. But the class template overloads the array indexing `operator[]` to enable reading and writing individual bits in the bitset as if they were elements of an array:

  ```
  buf[0] = 1;  // set least-significant bit to 1
  buf[2] = 1;  // set bit indexed 2 to 1
  int b = buf[7];  // access bit indexed 7 as an int
  ```

- See the documentation of bitset for more information about its API
Next time

- Graphs
- Vertices, edges, paths, cycles
- Sparse and dense graphs
- Representations: adjacency matrices and adjacency lists
- Implementation notes

Reading: Weiss, Chapter 9