Bitwise I/O

Step 1

We introduced the idea of **lossless data compression** with the goal of encoding data in such a way that we would be able to store data in smaller files on a computer with the ability to decode the encoded data perfectly, should we want to. Then, we discussed **Huffman Coding**, which allowed us to take a message and encode it in the fewest number of bits possible. However, now that we have this string of bits, how do we actually go about writing it to disk?

Recall that the smallest unit of data that can be written to disk is a *byte*, which is (for our purposes) a chunk of 8 *bits*. However, we want to be able to write our message to disk one bit at a time, which sounds impossible given what we just said. In this section, we will be discussing **Bitwise I/O**: the process of reading and writing data to and from disk *bit*-wise (as opposed to traditional I/O, which is *byte*-wise). The discussion will be using C++ specifically, but the process is largely the same in other languages.

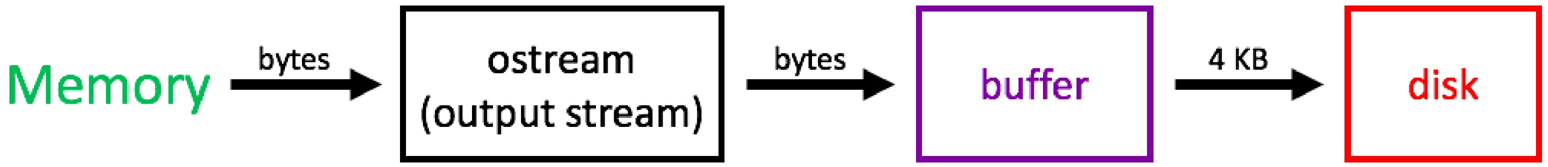
Step 2

In regular bytewise I/O, we like to think that we write a single byte to disk at a time. However, note that accessing the disk is an *extremely* slow operation. We won't go into the computer architecture details that explain *why* this is the case, but take our word for now that it is. As a result, instead of writing every single byte to disk one at a time (which would be very slow), computers (actually, programming languages) have a "write buffer": we designate numerous bytes to be written, but instead of being written to disk immediately, they are stored in a buffer in memory (which is fast), and once the buffer is full, the *entire buffer* is written to disk at once. Likewise, for *reading* from disk, the same logic holds: reading individual bytes is extremely slow, so instead, we have a "read buffer" in memory to which we read *multiple* bytes from disk, and we pull individual bytes from this buffer. Typically, a default value for a disk buffer is 4 kilobytes (i.e., 4,096 bytes).

The basic workflow for writing *bytes* to disk is as follows:

- Write bytes to buffer, one byte at a time
- Once the buffer is full, write the entire buffer to disk and clear the buffer (i.e., "flush" the buffer)
- Repeat

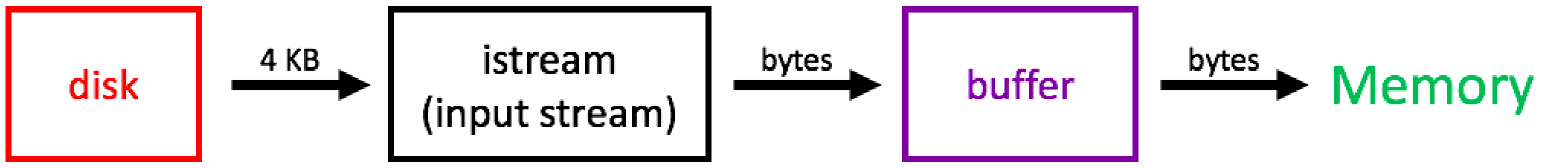
In most languages, we can create an "output stream" that handles filling the buffer, writing to disk, etc. for us on its own. Below is a typical workflow of **writing** some *bytewise* data from **memory** to **disk**:



Similarly, the basic workflow for reading *bytes* from disk is as follows:

- Read enough bytes from disk to fill the entire buffer (i.e., "fill" the buffer)
- Read bytes from buffer, one byte at a time, until the buffer is empty
- Repeat

Like with writing from disk, in most languages, one can create an "input stream" that handles reading from disk, filling the buffer, etc. Below is a typical workflow of **reading** some *bytewise* data from **disk** to **memory**:



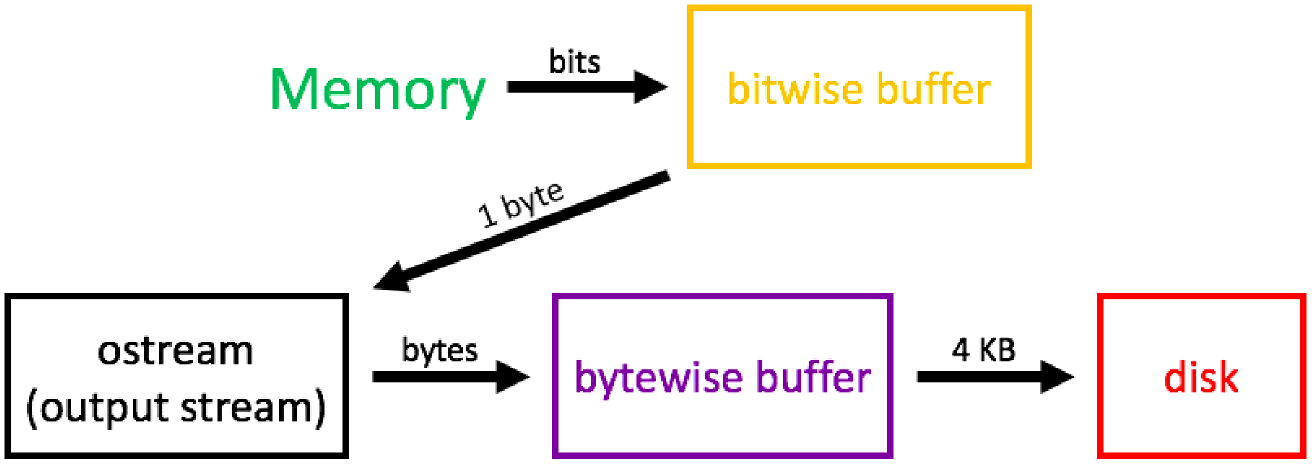
Step 3

We've seen a workflow for writing *bytes* to disk (as well as for reading bytes from disk), but we want to be able to read/write *bits*. It turns out that we can use the same idea with a small layer on top to achieve this: we can build our own *bitwise* buffer that connects with the existing *bytewise* buffer. We will design our bitwise buffer to be 1 byte (i.e., 8 bits).

The basic workflow for writing *bits* to disk is as follows:

- Write bits to bitwise buffer, one bit at a time
- Once the bitwise buffer is full, write the bitwise buffer (which is 1 byte) to the bytewise buffer and clear the bitwise buffer (i.e., "flush" the bitwise buffer)
- Repeat until the bytewise buffer is full
- Once the bytewise buffer is full, write the entire bytewise buffer to disk and clear the bytewise buffer (i.e., "flush" the bytewise buffer)
- Repeat from the beginning

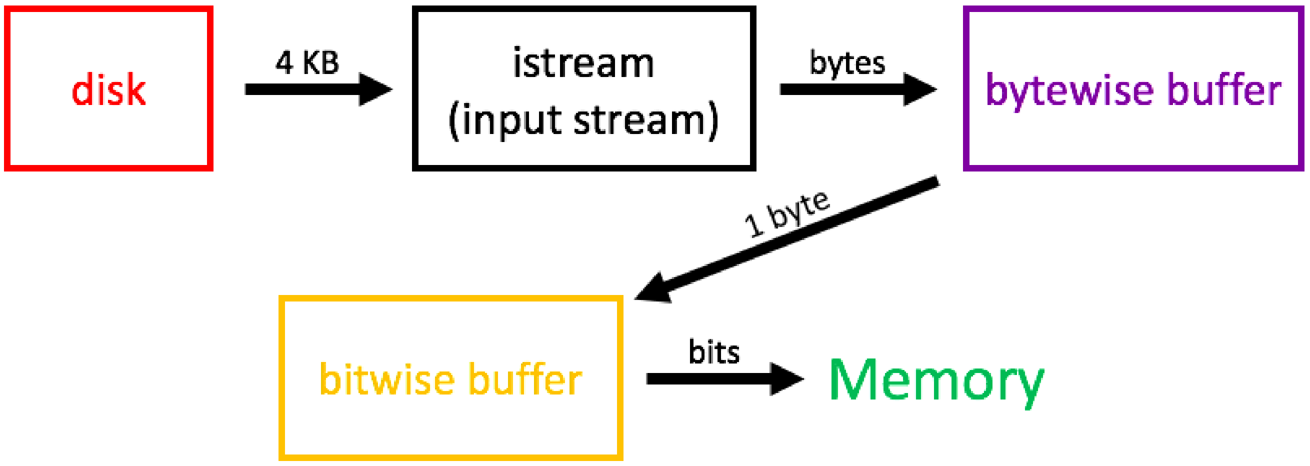
Below is a typical workflow of **writing** some *bitwise* data from **memory** to **disk**:



Similarly, the basic workflow for reading *bits* from disk is as follows:

- Read enough bytes from disk to fill the entire bytewise buffer (i.e., "fill" the bytewise buffer)
- Read 1 byte from the bytewise buffer to fill the bitwise buffer (i.e., "fill" the bitwise buffer)
- Read bits from bitwise buffer, one bit at a time
- Once the bitwise buffer is empty, read another byte from the bytewise buffer to fill the bitwise buffer, repeat (i.e., "fill the bitwise buffer)
- Once the bytewise buffer is empty, repeat from the beginning (i.e., "fill" the bytewise buffer)

Below is a typical workflow of **reading** some *bitwise* data from **disk** to **memory**:

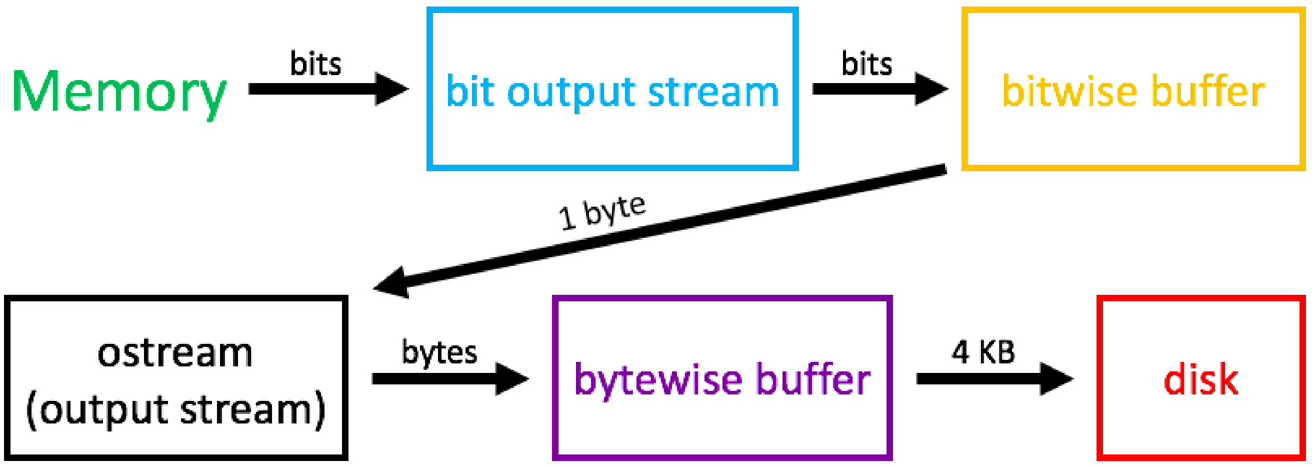


Step 4

Programming languages include "input streams" and "output streams" so that we as programmers don't have to worry with low-level details like handling bytewise buffers. With similar motivation, instead of us directly using a bitwise buffer, it would be better design if we were to design a "bit input stream" and a "bit output stream" to handle reading/writing the bitwise buffer for us automatically. Before we embark on the journey of actually designing the "bit input stream" and "bit output stream," let's reformulate our input and output workflows.

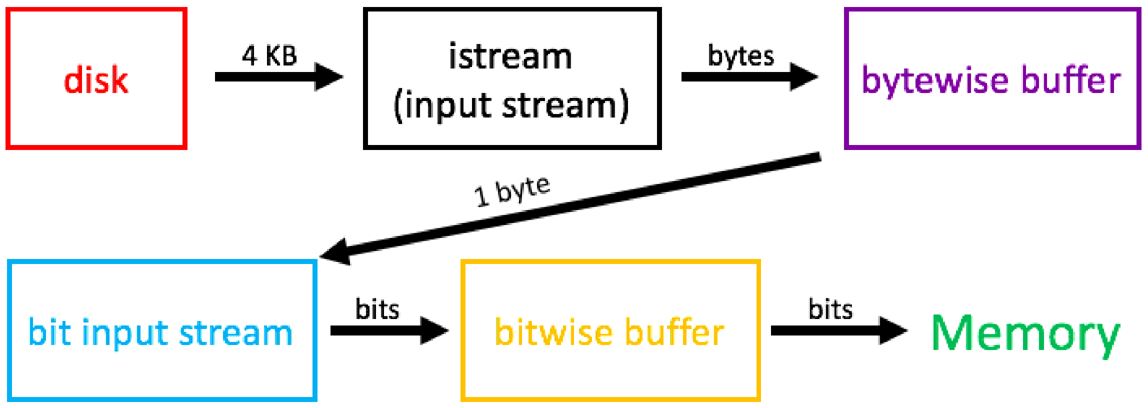
To write to disk, we will add a bit output stream to which we will write our bitwise data, and this bit output stream will feed into the regular output stream.

Below is the updated workflow of **writing** some *bitwise* data from **memory** to **disk**:



Similarly, to read from disk, we will add a bit input stream from which we will read our bitwise data, and this bit input stream will feed from the regular input stream.

Below is the updated workflow of **reading** some *bitwise* data from **disk** to **memory**:



Step 5

Thus far, we have mentioned that we flush the bytewise buffer once it's been filled, which means that we write all of the bytes to disk. What about if we have run out of bytes we want to write, but our bytewise buffer is not full? We mentioned previously that a "good size" for a bytewise buffer is 4 KB (i.e., 4,096 bytes), but what if we only have 100 bytes we wish to write to disk? The solution is trivial: we just flush the bytewise buffer by writing whatever bytes we *do* have to disk. In the example above, instead of writing 4 KB, we only write 100 bytes.

However, with a bitwise buffer, things are a bit more tricky if we run out of bits to write before the bitwise buffer is full. The difficulty is caused by the fact that the smallest unit that can be written to disk is 1 *byte*, which is 8 *bits*. For example, say we want to write to disk a sequence of bits that is not a multiple of 8 (e.g. 111111111111):

- We would write the first 8-bit chunk (11111111), and 1111 would be left
- We would then try to write the remaining chunk (1111), but we are unable to do so because it is not a full byte

The solution is actually still quite simple: we can simply "pad" the last chunk of bits with 0s such that it reaches the full 8 bits we need to write to disk. Because bytes and bits are traditionally read from "left" to "right" (technically the way it is laid out in the computer architecture does not have "left" or "right" directionality, but instead "small offset" to "large offset"), we want our padding 0s to be on the right side of our final byte such that the bits remain contiguous. In the same example as above:

- We would write the first 8-bit chunk (11111111), and 1111 would be left
- We would then pad 1111 to become 11110000, and we would write this padded byte
- The result on disk would be: 11111111   11110000 (the space implies that the bits are two separate bytes)

This way, when we read from disk, we can simply read the bits from left to right. However, note that we have no way of distinguishing the padding 0s from "true" bits. For example, what if we were to write the bit sequence 1111111111110000?

- We would write the first 8-bit chunk (11111111), and 11110000 would be left
- We would then write the remaining 8-bit chunk (11110000)
- The result on disk would be: 11111111   11110000 (the same two bytes written to disk as before)

As can be seen, because we cannot distinguish the padding 0s from "true" bits (as they're indistinguishable on disk), we need to implement some clever way of distinguishing between them manually. For example, we could add a "header" to the beginning of the file that tells us how many bits we should be reading. For example, we could do the following in the previous example:

- 00001100   11111111   11110000
- The first byte (00001100) tells us how many bits we should expect to read (00001100 = 12)
- We would then read the next byte (11111111) and know that we have read 8 bits (so we have 4 bits left to read)
- We would then read the next byte (11110000) and know that we should only read the first 4 bits (1111)

It is important to note that this exact header implementation ("How many bits should we expect to read?") is just *one* example. Thus, in whatever applications you may develop that require bitwise input and output, you should think of a clever header that would be optimal for your own purposes.

Step 6

As we have hinted at, our end-goal is to create two classes, **`BitOutputStream`** and **`BitInputStream`**, that will handle bitwise output and input for us. Specifically, our **`BitOutputStream`** class should have a **`writeBit()`** function (write a single bit to the bitwise buffer) and a **`flush()`** function (write the bitwise buffer to the output stream, and clear the bitwise buffer), and our **`BitInputStream`** class should have a **`readBit()`** function (read a single bit from the bitwise buffer) and a **`fill()`** function (read one byte from the input stream to fill the bitwise buffer).

Below is an example of a potential **`BitOutputStream`** class. Some of the details have been omitted for you to try to figure out.

```
class BitOutputStream {
    private:
        unsigned char buf; // bitwise buffer (one byte)
        int nbits;         // number of bits that have been written to the bitwise buffer
        ostream & out;     // reference to the bytewise output stream (a C++ ostream)

    public:
        // constructor: assign 'out' to 'os', 'buf' to 0, and 'nbits' to 0
        BitOutputStream(ostream & os) : out(os), buf(0), nbits(0) {}

        // send the bitwise buffer to the output stream, and clear the bitwise buffer
        void flush() {
            out.put(buf);  // write the bitwise buffer to the ostream
            out.flush();   // flush the ostream (optional, slower to do it here)
            buf = 0;       // clear the bitwise buffer
            nbits = 0;     // bitwise buffer is cleared, so there are 0 bits in it
        }

        // write bit to the bitwise buffer
        void writeBit(unsigned int bit) {
            // flush the bitwise buffer if it is full
            if(nbits == 8) {
                flush();
            }

            // set the next open bit of the bitwise buffer to 'bit' (how?)

            // increment the number of bits in our bitwise buffer
            nbits++;
        }
};
```

Below is an example of a potential **`BitInputStream`** class. Some of the details have been omitted for you to try to figure out.

```
class BitInputStream {
    private:
        unsigned char buf;  // bitwise buffer (one byte)
        int nbits;          // number of bits that have been read from bitwise buffer
        istream & in;       // reference to the bytewise output stream (a C++ ostream)

    public:
        // constructor: assign 'in' to 'is', 'buf' to 0, and 'nbits' to 8
        BitInputStream(istream & is) : in(is), buf(0), nbits(8) {}

        // fill the bitwise buffer by reading one byte from the input stream
        void fill() {
            buf = in.get(); // read one byte from istream to bitwise buffer
            nbits = 0;      // no bits have been read from bitwise buffer
        }

        // read bit from the bitwise buffer
        unsigned int readBit() {
            // fill bitwise buffer if there are no more unread bits
            if(nbits == 8) {
                fill();
            }

            // get the next unread bit from the bitwise buffer (how?)
            unsigned int nextBit = ??;

            // increment the number of bits we have read from the bitwise buffer
            nbits++;

            // return the bit we just read
            return nextBit;
        }
};
```

STOP and Think: In our `BitOutputStream` class, we flush the C++ `ostream` in our `flush()` function, but we mention that it is optional, and that it slows things down. Why would flushing the `ostream` here slow things down? Where might we want to flush the `ostream` instead to keep things as fast as possible?

Step 7

As you hopefully noticed in the previous step, we omitted some of the logic from `writeBit()` and `readBit()` for you to try to figure out on your own. Specifically, in both functions, we omitted bitwise operations that perform the relevant reading/writing of the bitwise buffer.

In the following steps, you will be solving some bitwise coding challenges that will hopefully push you towards the bitwise operation logic necessary to complete `writeBit()` and `readBit()`.

Step 8

Code Challenge: Setting the Rightmost Bit

You will need to fill in the body of `setRightBit()`. The input will be a byte (i.e., 8 bits) `input` and a single bit `bit`, and you will need to set the rightmost bit of `input` to `bit` and return the result.

For example, if `input` is 254 (i.e., 1111111**0**) and `bit` is 1, the result would be 255 (i.e., 1111111**1**).

Sample Input:

```
254 1
```

Sample Output:

```
255
```

To solve this problem please visit https://stepik.org/lesson/30030/step/8

Step 9

Code Challenge: Setting the Leftmost Bit

You will need to fill in the body of `setLeftBit()`. The input will be a byte (i.e., 8 bits) `input` and a single bit `bit`, and you will need to set the leftmost bit of `input` to `bit` and return the result.

For example, if `input` is 127 (i.e., **0**1111111) and `bit` is 1, the result would be 255 (i.e., **1**1111111).

Sample Input:

```
127 1
```

Sample Output:

```
255
```

To solve this problem please visit https://stepik.org/lesson/30030/step/9

Step 10

Code Challenge: Setting the *n*th Bit From the Right

You will need to fill in the body of `setBit()`. The input will be a byte (i.e., 8 bits) `input`, a single bit `bit`, and an integer `n`, and you will need to set the `n`-th bit of `input` (0-based counting from the right) to `bit` and return the result.

For example, if `input` is 85 (i.e., 0101**0**101), `n` is 3, and `bit` is 1, the result would be 93 (i.e., 0101**1**101).

Sample Input:

```
85 3 1
```

Sample Output:

```
93
```

To solve this problem please visit https://stepik.org/lesson/30030/step/10

Step 11

Code Challenge: Extracting the Rightmost Bit

You will need to fill in the body of `extractRightBit()`. The input will be a byte (i.e., 8 bits) `input`, and you will need to return the rightmost bit of `input` (either a 1 or a 0).

For example, if `input` is 255 (i.e., 1111111**1**), the result would be **1**.

Sample Input:

```
255
```

Sample Output:

```
1
```

To solve this problem please visit https://stepik.org/lesson/30030/step/11

Step 12

Code Challenge: Extracting the *n*th Bit From the Right

You will need to fill in the body of `extractBit()`. The input will be a byte (i.e., 8 bits) `input`, and an integer `n`, and you will need to return the `n`-th bit of `input` (0-based counting from the right) (either a 1 or a 0).

For example, if `input` is 247 (i.e., 1111**0**111) and `n` is 3, the result would be **0**.

Sample Input:

```
247 3
```

Sample Output:

```
0
```

To solve this problem please visit https://stepik.org/lesson/30030/step/12