Problem Set 8

Due before midnight on Monday, December 5, 2016.

1 Assignment Goals

1. Learn low-level bit representations of data.
2. Get greater insight into how standard I/O streams work.
3. Use non-polymorphic derivation to eliminate duplicate code.

2 Bit Strings

We are used to working with files that consist of a sequence of 8-bit *bytes*. The standard `iostream` class and its cousins, `ifstream` and `ofstream`, allow for easy processing of byte sequences.

Some applications work instead with *binary* data – sequences of `bits` where the underlying byte structure may have little semantic relevance. Binary files are generally uningelligable to a human just looking at them. Typical examples of binary files are executable machine code, encrypted data, and compressed data, images, video, and sound files.

Abstractly, bitstrings consist of an *arbitrary*-lengh sequence of *bits*, just as ordinary text files consist of an arbitrary-length sequence of bytes. However, modern computers are based on bytes, and bitstrings are generally stored as sequences of bytes, where each 8-bit block of bits is packed into a single byte. As a consequence, the only bitstrings that can be directly represented are those whose length (in bits) is a multiple of eight.

Applications that need to store arbitrary length bitstrings must encode their data as bytes on writing to a file and decode them back to the original bitstring on reading. Such encoding considerations are generally orthogonal to the application and can greatly complicate the code. Good object-oriented design dictates that independent modules should be in separate classes with clean interfaces. The application that wants to use bit streams should invoke bit stream classes, and the conversion between bit streams and byte streams should be relegated to separate classes.

3 Problem

In this assignment, you will implement the following three classes for handling bit stream files.

1. `BStreamBase` is the base class for both of the following classes.
2. `BIFStream` supports reading a file-encoded bit stream.
3. `BOFStream` supports writing a file-encoded bit stream.

Your classes will present interfaces to the client programs that is very similar to the standard streams interfaces with which you are familiar. They will read and write files in a special file format, described below, which I call *packed bit stream* files. By convention, packed bit stream files will have names with a `.pbs` suffix.

3.1 Backed Bit Stream File Format

I now describe the format of a `.pbs` file. Let N be the length of the file in bytes. Assume $N \geq 2$. Then each of the first $N-2$ bytes contain eight bits from the bit stream, packed left to right in the byte. Thus, the first of those eight bits appears in the high-order bit position of the byte and the last appears in the low-order bit position.

The next-to-last byte of the file contains the last k bits from the bit stream, where $1 \leq k \leq 8$. These bits are *right-justified* within the byte, and the remaining *padding* bits must all be 0's. The last byte of the file is called the *count* byte and contains the binary representation of the number k.

The case $N = 1$ contains only a count byte, which must be 0. It represents the empty bit stream, that is, the bit stream consisting of no bits.

Every bit stream can be represented uniquely as a `.pbs` file. However, not every file is a valid encoding of a bit stream. In particular, the file must not be empty itself, the count byte must be 0 if the file length is 1 and between 1 and 8 otherwise. Moreover, if the next-to-last byte has only $k < 8$ bits, the leftmost $8-k$ padding bits must all be zero.

A few examples should help make this clear.

Bit Stream	Byte Representation	Notes
(empty)	`0x00`	special case – empty bit stream
101	`0x05 0x03`	three bits are right-justified
00101	`0x05 0x05`	five bits are right-justified
10100001	`0xa1 0x08`	last byte full; no padding
1111000011011	`0xf0 0x1b 0x05`	13-bits in two bytes
110	`0xa6 0x03`	illegal; padding bits are not 0
10100001	`0xa1 0x00 0x00`	illegal; count is 0 for non-empty bit stream
1101	`0x0d 0x14`	illegal; count is out of range

3.2 Bit Stream Base Class

Bit stream objects should look as much like ordinary I/O streams as possible. In particular, they should support similar error states as the standard `ios` class. Namely, there are three error flags packed into a single `int` variable `state`: badflag, eofflag, and failflag. Badflag is in bit position 0 (the low-order bit of `state`), eofflag is in position 1, and failflag is in position 2. Thus, state 6 means the eofflag and failflag are both set. State 0 is the good state.

To conveniently manipulate I/O states, you should define static int constants `goodbit=0`, `badbit=1`, `eofbit=2`, and `failbit=4`. With these constants, one can use the bitwise "or" operator `'|'` to form the state. Thus, state 6 is equal to `eofbit|failbit`. Similarly, to test if the eofflag is set, one can test `(state & eofbit)!=0`.

`BStreamBase` supports those parts of the bit stream that are common to both input and output bit streams. In particular, it contains `state` as a protected data member. Public functions are

- `rdstate()` returns the state as an `int`.
- `clear(int s=goodbit)` sets the state to `s`. Just like the standard `clear()` function, if called with no arguments, it clears all state bits.
- `setstate( int s )` does a bitwise "or" of the current state with `s` and stores the result back in `state`. For example, `setstate( eofbit )` turns on the eofflag without changing the other flags.
- The Boolean state functions `good()`, `eof()`, `fail()`, and `bad()` are defined exactly the same as for standard streams. For example, `fail()` returns `true` if either badflag or failflag are set.

3.3 Bit Stream Derived Classes

`BIFStream` is publicly derived from `BStreamBase`. It's constructor takes a `const string` parameter that gives the name of the `.pbs` file to open for reading when `BIFStream` is instantiated. It's destructor calls the bit stream's `close()` function. Other public functions are `bool is_open()`, `void close()`, and `unsigned char getBit()`.

`getBit()` returns the next bit from the bit stream, where bit 0 is represented by the byte `0x00` and bit 1 is represented by `0x01`. It should set badflag if the underlying file violates the file format described above. It should set eofflag and failflag if it is unable to return a bit because there are no more bits remaining in the bit stream.

`BOFStream` is publicly derived from `BStreamBase`. It's constructor takes a `const string` parameter that gives the name of the `.pbs` file to open for writing when `BOFStream` is instantiated. The constructor also takes an optional parameter of type `ios::openmode` that gives the mode in which to open the underlying output stream. It defaults to `ios::out`. The destructor calls the bit stream's `close()` function. Other public functions it supports are `bool is_open()`, `void close()`, `void putBit( int b )` and `void putByte( unsigned char newChar )`.

The argument to `putBit()` should be the integer 0 or 1, which it appends to the output bit stream. The argument to `putByte()` should be an unsigned character `newChar`. It writes all 8 bits of `newChar` to the output bit stream. While it could simply extract the 8 bits one at a time and call `putBit()` on each, it can be implemented more efficiently with shifting and masking. In both cases, badflag and failflag should be set if there is any error writing to the underlying output stream.

4 Testing

In order to test your program, you should write two commands `packer` and `unpacker` that convert between bit streams as described above and strings of `'0'` and `'1'` characters. Both commands take two filename arguments.

The command

```
> packer in.txt out.pbs
```

reads a file `in.txt` consisting of `'0'` and `'1'` characters, interprets them as bits 0 and 1 respectively, and writes them to the bit stream file `out.pbs`. The command should ignore whitespace on input but otherwise check that the only non-whitespace characters present are `'0'` and `'1'`.

The command

```
> unpacker in.pbs out.txt
```

does the opposite. It reads the packed bit stream file `in.pbs` and writes the corresponding characters `'0'` and `'1'` to file `out.txt`. For readability, a newline character should be inserted after every eight digits of output. Also, make sure that the last line is also terminated by a newline, even if it is shorter than 8.

`packer` and `unpacker` should use the public functions `good()`, `fail()`, etc., to check for errors and end-of-file as appropriate after each bit stream and text stream operation.

5 Programming Hints

Both input and output bit streams should maintain a single byte buffer along with a length variable that says how many unread bits are in the buffer (for input), or how many bits are in the buffer

waiting to be written out (for output).

The tricky part of this assignment is in properly handling end of file. For input, one needs to keep a 2-byte lookahead buffer in order to determine whether a newly-read byte is a data byte or the count byte, and also to know how many bits in that byte are valid.

In general, byte k of the input file will be in the byte buffer, and bytes $k + 1$ and $k + 2$ will be in the lookahead buffer. When the byte buffer becomes empty, the first lookahead byte is move into the byte buffer, the second lookahead byte becomes the first, and you attempt to read a new byte into the second slot of the lookahead buffer.

If you fail to read a new byte because of end-of-file, you know that the first byte in the lookahead buffer is the count byte. You continue processing until the byte buffer becomes empty, at which time your function `eof()` reports end-of-file on the bit stream.

For bit stream output, the bit stream `close()` function must write the byte buffer to the output stream (if non-empty), followed by the count. In case the byte buffer is empty, the count should be 8 to reflect the fact that the previously-written byte was full. However, the empty file must be treated as a special case since then the byte buffer is empty but no previous byte was written. In this one case, the count byte should be 0, not 8.

6 Grading Rubric

Your assignment will be graded according to the scale given in Figure 1.

#	Pts.	Item
1.	1	A well-formed `Makefile` or `makefile` is submitted that specifies compiler options `-O1 -g -Wall -std=c++11`. Running `make` successfully compiles and links the project and results in two executable files, `packer` and `unpacker`.
2.	6	`packer` successfully packs all of the furnished `.txt` files or correctly detects and reports error conditions.
3.	6	`unpacker` successfully unpacks all of the furnished `.pbs` files or correctly detects and reports error conditions.
4.	6	All of the instructions in sections 3, 4, and 5 are followed.
5.	1	All relevant standards from previous problem sets are followed regarding good coding style, submission, identification of authorship on all files, and so forth.
	20	Total points.

Figure 1: Grading rubric.