Problem Set 9

Due before midnight on Monday, December 12, 2016.
No late papers will be accepted after Thursday, December 15, the last day of Reading Period.

1 Assignment Goals

This is a short half-credit assignment that builds on Problem Set 8. The goals are:

1. See an application for the bit streams of Problem Set 8.
2. Learn about prefix codes.
3. Learn how to use the standard container `map()`.
4. Gain familiarity with different data representations.

2 Codes

A *code* is a function from a set of *source symbols* to a set of bit strings called *codewords*. For example, the ASCII code maps characters to 8-bit strings.

A *variable-length code* is a code whose codewords are not all the same length. A *prefix code* is a code such that no codeword is the prefix of another (longer) code word.

A code is often presented as a *code table*. The left column gives the source symbol; the right column gives its codeword.

Symbol	Codeword
a	01
c	101
t	11

To encode a sequence of symbols, the code replaces each symbol with its codeword. Using the above example, the encoding of "cat" would be 1010111.

Prefix codes are self-delimiting in the sense that an encoding can be uniquely parsed into it constituent code words. The decoding process finds the longest prefix of the encoding that is a codeword. Given the encoding 1010111, we begin reading it a bit at a time, looking for a codeword. 1 is not a codeword. 10 is not a codeword. 101 is the codeword corresponding to 'c'. By the prefix property, no other codeword begins with 101, so we have found the correct decoding of the first message symbol, which is 'c'. We discard 101 from the encoding and continue the process on the remainder 0111. We see that 01 is the code for 'a', output it, and continue with 11, which is the code for 't'.

You can see that decoding is considerably more complicated than encoding. For this assignment, you will only be doing the encoding part.

3 Problem

Write a command `encode` that takes three command line arguments:

1. `code table` is an input file that contains the code table for a prefix code.
2. `message` is an input text file that contains a message to be encoded.
3. `encoded` is an output `.pbs` bit stream file as described in problem set 8 that will receive the encoded message.

Your program should construct a `map<const unsigned char, string>` to represent the code table. Each line of the code table file contains two whitespace-delimited fields. The first field is a base-16 (hex) number describing a single byte `b`. The second field is a string of digits `'0'` and `'1'`. It should be read into a string variable `s`. The pair (b, s) should then be put into the map

For example, the code table above would be represented by the file

```
61 01
63 101
74 11
```

Note that $(61)_{16}$ is the hex code for 'a'.

When the code table has been read in and the map constructed, the map should be printed to `cout` as a check that the map was successfully constructed.

Your program should then proceed to encode the message. The message should be read one byte at a time. Each byte should be looked up in the code table using `map::find()`. The corresponding code word should then be written to the output `BOFStream`, one bit at a time like you did for the `packer` command of the last problem set.

As always, you should check for obvious errors and report them by throwing `Fatal()`.

I will supply some sample files so that you can check your output. If I have time, I will also supply a command file `decode` to map encoded files back to text files. But you can get started just using the sample files.

4 Programming Notes

In constructing the map, you will need to read the first field of the code table file as a hex integer and then convert it to an `unsigned char`. To read a hex number from stream `in` into an `int` variable `b`, you can use `in >> hex >> b;`. To convert `b` to an `unsigned char`, you can write `ch = b;`, where `ch` has type `unsigned char`. However, the conversion happens implicitly if you use `b` where an `unsigned char` is needed, such as in the map you will be constructing.

The easiest way to put the pair (b, s) into the map is to use `map::emplace()`.[1] To search the map `ct` for key `b`, write `it=ct.find(b)`.[2] Here, `it` is a map iterator of type `map<const unsigned char,string>::iterator`. If `find()` fails to find its search key, it returns the map iterator `end()`. Thus, `it==ct.end()` is true if the search failed.

Finally, to print the map, you can use the range-based iterator to go through the map, e.g., `for (pair<const unsigned char, string> p : ct)`. Here, `p` is successively bound to each pair in the map. The first and second elements of the pair can be obtained as `p.first` and `p.second`, respectively. As usual, to print an `unsigned char` as a hex number, you will need to cast it to an `int` and use the `hex` manipulator.

[1] See http://www.cplusplus.com/reference/map/map/emplace/.
[2] See http://www.cplusplus.com/reference/map/map/find/.

5 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 an executable file, `encode`.
2.	4	`encode` successfully reads the code table into a `map` and prints the map to `cout`.
3.	4	`encode` successfully encodes the message.
4.	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.