










# CS 106B: Programming Abstractions in C++

## Summer 2014





**CS 106B**

 [Home](#)  
 [Textbook](#)  
 [FAQ](#) /  [Links](#)  
 [Handouts](#)

**Course Info**

 [Lectures](#)  
 [Homework](#)  
 [Sections](#)  
 [Exams](#)

**Getting Help**

 [Staff/SLs](#)  
 [LaIR Hours](#)  
 [Message Forum](#)  
 [Work@Home](#)

**Documentation**

 [Stanford C++ Lib](#)  
 [CppReference.com](#)  
 [CPlusPlus.com](#)  
 [106B Style Guide](#)

## Homework 6 (Huffman Encoding) FAQ

**Q: I don't understand what is going on in this assignment.**

A: Take a look at the pictures in the assignment writeup and lecture slides. They explain how the priority queue works with the algorithm we've given you. We also highly recommend that you read the Supplemental handout on Huffman encoding, posted on the Homework page next to the Huffman Encoding spec document. The recent section on binary trees may also help.

**Q: The spec says I am not supposed to modify the .h files. But I want to use a helper function. Don't I need to modify the .h file to add a function prototype declaration for my helpers? Can I still use helper functions even if I don't modify the .h file?**

A: Do not modify the provided .h file. Just declare your function prototypes in your .cpp file (near the top, above any code that tries to call those functions) and it'll work fine. You can declare a function prototype anywhere: in a .cpp file, in a .h file, wherever you want. The idea of putting them in a .h file is just a convention. When you `#include` a file, the compiler literally just copy/pastes the contents of that file into the current file. We have already done this on hw1, hw2, and others.

**Q: In Part 1 of encoding, what is a "pseudo EOF"? How do I add a "pseudo EOF" to a map?**

A: `PSEUDO_EOF` is a global constant that is visible to your program. It is just an `int` constant whose value happens to be 256, so you can put it in your map as a key with the value of 1. Something like this:

```
myMap.put(PSEUDO_EOF, 1);
```

You also need to explicitly write out a single occurrence of `PSEUDO_EOF`'s binary encoding when you compress a file, in Step 4 (the actual encoding of the data, represented by the `encodeData` function). Write out all of the necessary bits to encode the file's data, and then after that, look up the binary encoding for `PSEUDO_EOF` and write out all of that encoding's bits to the file at the end.

**Q: What is the difference between a "pseudo EOF" and a "real" EOF? What is the value of "real" EOF? Is it -1? Because file input functions like `get()` return -1 when you reach the end of the file, so are they returning "real" EOF?**

There is a difference between `PSEUDO_EOF` and the notion of a "real" EOF. `PSEUDO_EOF` is 256, and it's a fake value that our program is using to signal the end of compressed data in a file. A real EOF is not -1. It is not a character or integer value at all; it is something decided internally by the operating system. The real file system knows where the

end of a file is because there is master table of data about all the files on the disk, and that table stores every file's length in bytes. The OS doesn't insert any special character at the end of each file; it just knows that you have hit the end-of-file once you have read a certain number of bytes equal to that file's length. The input stream's `get` function just returns `-1` when you're done because that's how they chose to indicate to you that the file was ended, not because an actual `-1` is on the hard disk.

**Q: What is `NOT_A_CHAR`? When will I see it? What do I need to use it for?**

A: `NOT_A_CHAR`, like `PSEUDO_EOF`, is a global constant that is visible to your program. It is just an `int`, so you can use it in places where a character is expected. The only place `NOT_A_CHAR` should be used in this assignment is when you create a `HuffmanNode` that has children, when you are combining nodes during Step 2 of the encoding process. The parent node has two subtrees under it and it doesn't directly represent any one character, so you store `NOT_A_CHAR` as the character data field of the parent node. That should be the only time you see `NOT_A_CHAR` and the only place you need to use it. You'll never see that value in an input or output file or anything like that.

**Q: In Part 2 of encoding, my tree doesn't get created correctly. How can I tell what's going on?**

A: We suggest inserting print statements in the function that builds the tree. The `HuffmanNodes` have a `<<` operator, so you can print them out. There is also a `printSideways` function provided that takes a `HuffmanNode*` and prints that entire tree sideways.

**Q: In Part 2 of encoding, my tree doesn't get created correctly. How can I tell what's going on?**

A: We suggest inserting print statements in the function that builds the tree. The `HuffmanNodes` have a `<<` operator, so you can print them out. There is also a `printSideways` function provided that takes a `HuffmanNode*` and prints that entire tree sideways.

**Q: In Part 2 of encoding, the contents of my priority queue don't seem to be in sorted order. Why?**

A: A `PriorityQueue`'s ordering is based on the priorities you pass in when you enqueue each element. Are you sure you are adding each node with the right priority?

**Q: In Part 2 of encoding, what should the priority queue's ordering be if the two nodes' frequencies are equal?**

A: If the counts are the same, just add them both with the same priority and let the priority queue decide how to relatively order those two items.

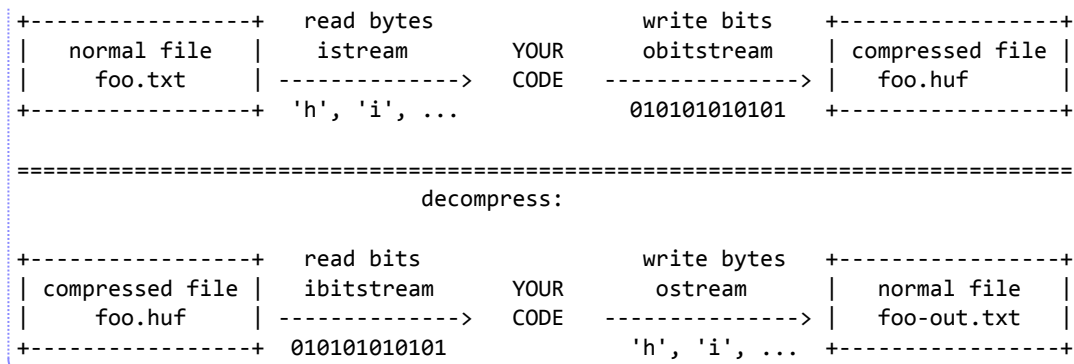
**Q: I don't understand the different kinds of input/output streams in the assignment. Which kind of stream is used in what situation? How do I create and initialize a stream? When do I open/close them?**

A: Here's a rundown of the different types of streams:

- An `istream` (aka `ifstream`) reads bytes from a file. You'd use this to read a normal file byte-by-byte so that you can compress its contents.
- An `ostream` (aka `ofstream`) writes bytes to a file. You'd use this to write to an uncompressed file byte-by-byte when you are decompressing.
- An `ibitstream` reads *bits* from a file. You'd use this to read a compressed file bit-by-bit when you are decompressing it.
- An `obitstream` writes *bits* to a file. You'd use this to write to a compressed file bit-by-bit when you are compressing.

Here's a diagram summarizing the streams:

compress:



You never need to create or initialize a stream; the client code does that for you. You are passed a stream that is ready to use; you don't need to create it or open it or close it.

**Q: How can I tell what bits are getting written to my compressed file?**

A: The main testing program has a "binary file viewer" option to print out the bits of a binary file. Between that and print statements in your own code for debugging, you should be able to figure out what bits came from where.

**Q: I don't understand the "header" for compress/decompress. How do I write the frequency table into the start of the binary file?**

A: Just use the << and >> operators to write your map into the stream, and then after that, read or write the binary bits as appropriate. Something like this:

```
// compress
output << frequencyTable; // write header
while (...) {
    output.writeBit(...); // write compressed binary data
}

// decompress
Map<int, int> frequencyTable;
input >> frequencyTable; // read header
while (...) {
    input.readBit(...); // read compressed binary data
}
```

**Q: What parts of the program need to worry about the header?**

A: Only compress and decompress. The other functions, such as encodeData and decodeData, should not worry about headers at all and should not have any code related to headers.

**Q: My individual step functions (buildFrequencyTable, encodeData, etc.) work fine, but my compress function always produces an empty file or a very small file. Why?**

A: Maybe you are forgetting to "rewind" the input stream. Your compress function reads over the input stream data twice: once to count the characters for the frequency table, and a second time to actually compress it using your encoding map. Between those two actions, you must rewind the input stream by writing code such as:

```
input.clear(); // removes any current eof/failure flags
input.seekg(0, ios::beg); // tells the stream to seek back to the beginning
```

**Q: Why do I have some unexpected junk characters at the end of my output when decoding?**

A: You need to look for the PSEUDO\_EOF as a marker to tell you when to stop reading. Make sure you insert a

PSEUDO\_EOF at the end of the output when you are encoding data. And make sure to check for PSEUDO\_EOF when decoding later.

**Q: My program works for most files, but when I try to decompress a big file like hamlet.txt, I get a crash. Why?**

A: It's possible that your algorithm is nesting too many recursive calls. Once you are done making one recursive walk down the tree, you should let the call stack unwind rather than making another recursive call to get back to the top of the tree.

**Q: My program works fine on text files, but it produces corrupt results for binary files like images (bmp, jpg) or sound files (mp3, wav). Why?**

A: This most commonly occurs when you store bytes from a file as type `char` rather than as type `int`. Use `int`. Type `char` works fine for ASCII characters but not for extended byte values that commonly occur in binary files.

**Q: My program runs really slowly on large files like hamlet.txt. How can I speed it up?**

A: It is expected that the code will take a little while to run on a large file. Our solution takes a few seconds to process Hamlet. Your program also might be slow because you're running it on a slow disk drive such as a USB thumb drive.

**Q: What should it do if the file to compress/decompress is empty?**

A: Your program should be able to handle this case. You'll write a header containing only the pseudo-EOF encoding, so the 0-byte file increases back up to around 7 bytes. When you decompress the file, it'll go back to being a 0-byte file. You may not even need to write any special code to handle the empty file case; it will "just work" if you follow the other algorithms properly.

**Q: What is the default value for a char? What char value can I use to represent nothing, or the lack of a character?**

A: The default char value is `'\0'`, sometimes called the 'null character'. (Not the same as `NULL`, the null pointer.) But Huffman nodes that have children should store `NOT_A_CHAR`, a constant declared by our support code.

**Q: When do I need to call my own freeTree function? Do I ever need to call it myself?**

A: If you ever create an encoding tree yourself as a helper to assist you in solving some larger task, then you should free that tree so that you don't leak memory. So for example, your `buildEncodingTree` function should *not* free the tree because it is supposed to return that tree to the client, and presumably that client will later free it. But if you call `buildEncodingTree` somewhere in your code because you want to use an encoding tree to help you, then when you are done using it, you should immediately call `freeTree` on it.

This document and its content are copyright © Cynthia Lee and Marty Stepp, 2014. All rights reserved. Any redistribution, reproduction, transmission, or storage of part or all of the contents in any form is prohibited without the authors' expressed written permission.