```
#pragma once

#include <map>
#include <string>
#include <vector>


// All functions used for lzw project part 1
std::map<std::string, int> compression_dictionary();
std::map<int, std::string> decompression_dictionary();
std::vector<int> compress(const std::string &);
std::string convert_to_bytes(std::string &);
std::vector<int> separate(std::string &, int);
std::string make_string(std::string &);
std::string decompress(std::vector<int> &);
std::string int_to_binary_string(std::vector<int>, std::string);
int binary_string_to_int(std::string);
```

```
#include "lzw435.hpp"
#include <bitset>
#include <cassert>
#include <iostream>
#include <iterator>
#include <sys/stat.h>

// Builds a dictionary of extended ASCII characters
// The pairs are (string, int) pairs
// These fill up the keys from 0 to 255
std::map<std::string, int> compression_dictionary() {

  std::map<std::string, int> c_dictionary;

  // Build the dictionary.
  auto dictionary_size = 256;
  for (auto x = 0; x < dictionary_size; ++x)
    c_dictionary[std::string(1, x)] = x;

  return c_dictionary;
}

// Builds a dictionary of extended ASCII characters
// The pairs are (int, string) pairs
// These fill up the keys from 0 to 255
std::map<int, std::string> decompression_dictionary() {
  std::map<int, std::string> d_dictionary;

  // Build the dictionary.
  auto dictionary_size = 256;
  for (auto x = 0; x < dictionary_size; ++x)
    d_dictionary[x] = std::string(1, x);

  return d_dictionary;
}

// Take a string and create a vector of integers to represent
// tokens in the string
std::vector<int> compress(const std::string &uncompressed) {

  std::map<std::string, int> dictionary =
      compression_dictionary(); // initialize dictionary

  std::string lookahead;
  std::vector<int> v;
  for (auto it = uncompressed.begin(); it != uncompressed.end(); ++it) {
    char current = *it;
    std::string consume = lookahead + current;

    // The entry is already in the dictionary
    if (dictionary.count(consume)) {
      lookahead = consume;
    } else { // lookahead is not in the dictionary, add it
      v.push_back(dictionary[lookahead]);

      // TODO: modify so that the rest of the file gets put into dictionary
      if (dictionary.size() == 4096)
        return v;

      // Add consume to the dictionary
      dictionary[consume] = dictionary.size() - 1;
      lookahead = std::string(1, current);
    }
  }
  if (!lookahead.empty())
    v.push_back(dictionary[lookahead]);

  return v;
}

/*
  Takes a string and reads 8 characters at a time.
  Converts the segments read into a character.
  Replaces each segment in the string with a character.
  Returns the string once the input string has been
  read to the end.
*/
std::string convert_to_bytes(std::string &s) {
  const int byte_size = 8;

  for (auto i = 0; i < s.size(); ++i) {
    std::string segments = s.substr(i, byte_size);
    int new_char = binary_string_to_int(segments);
    s.replace(i, byte_size, 1, static_cast<char>(new_char));
  }
  return s;
}

std::string make_string(std::string &s) {
  std::string str;
  for (auto i = 0; i < s.length(); ++i) {
    std::bitset<8> b(static_cast<int>(s.at(i)));
    str.append(b.to_string());
  }

  return str;
}

/*
  Takes a string and bit_length
  Creates strings the length of bit_length, then calls
  binary_string_to_int to convert the string to an integer.
  This integer is then added to a vector.
  Returns the vector after the entire input string has been
  separated into substrings and the integer result of each string
  has been computed.
*/
std::vector<int> separate(std::string &s, int bit_length) {
  std::vector<int> v;

  int zeros = bit_length - 8;

  assert((s.length() - zeros) % bit_length == 0);

  for (int i = 0; i < s.size() - zeros; i += bit_length) {
    std::string str = s.substr(i, bit_length);
    assert(str.length() == bit_length);
    v.push_back(binary_string_to_int(str));
  }
  return v;
}

/*
  Takes a vector of integers and returns a string
```

```
  representing each integer as a string in the map.
  It builds the dictionary and then recursively computes the
  value of the string as an integer, then checks
  if the value is in the dictionary. If it is not, it adds it.
  The resulting string is expected to be the file contents
  of the original file before compression.
*/
std::string decompress(std::vector<int> &v) {
  assert(!v.empty());
  std::map<int, std::string> dictionary = decompression_dictionary();

  std::string lookahead(1, static_cast<char>(v.at(0)));
  std::string result = lookahead;
  std::string entry;
  for (auto i = 1; i < v.size(); ++i) {
    int value = v.at(i); // value at i in vector

    // If the value is in the dictionary
    if (dictionary.count(value)) {
      entry = dictionary[value]; // set entry to the string pair of value
    } else if (value == dictionary.size()) {
      entry = lookahead + lookahead.at(0);
    } else
      throw "Cannot decompress!\n";

    result.append(entry);

    // Add the entry to the dictionary
    if (dictionary.size() < 4096)
      dictionary[dictionary.size()] = lookahead + entry.at(0);

    lookahead = entry;
  }
  return result;
}

std::string int_to_binary_string(std::vector<int> v, std::string s) {

  while (!v.empty()) {
    const int bits = 12;
    std::bitset<bits> b(v.front());

    s.append(b.to_string());
    v.erase(v.begin());
  }

  assert(v.empty());

  if (v.empty())
    return s;
}

int binary_string_to_int(std::string s) {

  int code = 0;
  if (s.size() > 0) {
    if (s.at(0) == '1')
      code = 1;
    s = s.substr(1);
    while (s.size() > 0) {
      code = code << 1;
      if (s.at(0) == '1')
```

```
        ++code;
      s = s.substr(1);
    }
  }
  return code;
}
```

```
#pragma once

#include <map>
#include <string>
#include <vector>
#include <tuple>


// All functions used for lzw project part 1
std::map<std::string, int> compression_dictionary();
std::map<int, std::string> decompression_dictionary();
std::vector<int> compress(const std::string &);
std::string convert_to_bytes(std::string &);
std::vector<int> separate(std::string &, int);
std::string make_string(std::string &);
std::string decompress(std::vector<int> &);
std::tuple<int, int> get_code_and_length(std::vector<int>);
std::string int_to_binary_string(std::vector<int>, std::string);
int binary_string_to_int(std::string);
```

```
#include "lzw435M.hpp"
#include <bitset>
#include <cassert>
#include <fstream>
#include <iostream>
#include <iterator>

// Builds a dictionary of extended ASCII characters
// The pairs are (string, int) pairs
// These fill up the keys from 0 to 255
std::map<std::string, int> compression_dictionary() {

  std::map<std::string, int> c_dictionary;
  // Build the dictionary.
  auto dictionary_size = 256;
  for (auto x = 0; x < dictionary_size; ++x)
    c_dictionary[std::string(1, x)] = x;

  return c_dictionary;
}

// Builds a dictionary of extended ASCII characters
// The pairs are (int, string) pairs
// These fill up the keys from 0 to 255
std::map<int, std::string> decompression_dictionary() {
  std::map<int, std::string> d_dictionary;
  // Build the dictionary.
  auto dictionary_size = 256;
  for (auto x = 0; x < dictionary_size; ++x)
    d_dictionary[x] = std::string(1, x);

  return d_dictionary;
}

// Take a string and create a vector of integers to represent
// tokens in the string
std::vector<int> compress(const std::string &uncompressed) {

  std::map<std::string, int> dictionary =
      compression_dictionary(); // initialize dictionary

  std::string lookahead;
  std::vector<int> v;
  for (auto it = uncompressed.begin(); it != uncompressed.end(); ++it) {
    char current = *it;
    std::string consume = lookahead + current;

    // The entry is already in the dictionary
    if (dictionary.count(consume)) {
      lookahead = consume;
    } else { // lookahead is not in the dictionary, add it
      v.push_back(dictionary[lookahead]);

      // TODO: modify so that the rest of the file gets put into dictionary
      if (dictionary.size() == 4096)
        return v;

      // Add consume to the dictionary
      dictionary[consume] = dictionary.size() - 1;
      lookahead = std::string(1, current);
    }
  }
```

```
  if (!lookahead.empty())
    v.push_back(dictionary[lookahead]);

  return v;
}

/*
  Takes a string and bit_length
  Creates strings the length of bit_length and adds
  them to a vector. Returns the vector after the entire
  input string has been separated into substrings.
*/
std::vector<std::string> separate(std::string &s) {
  std::vector<std::string> v;
  for (auto i = 0; i < s.size(); ++i) {
    std::string str = s.substr(s.at(i), s.at(i + bit_length));
    std::cout << str;
    // assert(str.length() == bit_length);
    v.push_back(str);
  }
  return v;
}

/*
  Takes a vector of integers and returns a string
  representing each integer as a string in the map.
  It builds the dictionary and then recursively computes the
  value of the string as an integer, then checks
  if the value is in the dictionary. If it is not, it adds it.
  The resulting string is expected to be the file contents
  of the original file before compression.
*/
std::string decompress(std::vector<int> &v) {
  assert(!v.empty());
  std::map<int, std::string> dictionary = decompression_dictionary();

  std::string lookahead(1, static_cast<char>(v.at(0)));
  std::string result = lookahead;
  std::string entry;
  for (auto i = 1; i < v.size(); ++i) {
    int value = v.at(i); // value at i in vector

    // If the value is in the dictionary
    if (dictionary.count(value)) {
      entry = dictionary[value]; // set entry to the string pair of value
    } else if (value == dictionary.size()) {
      entry = lookahead + lookahead.at(0);
    } else
      throw "Cannot decompress!\n";

    result.append(entry);

    // Add the entry to the dictionary
    if (dictionary.size() < 4096)
      dictionary[dictionary.size()] = lookahead + entry.at(0);

    lookahead = entry;
  }
  return result;
}

std::tuple<int, int> get_code_and_length(std::vector<int> v) {
  // Return binary string of values in vector
  if (v.empty())
    throw "ERROR: Cannot compute an empty vector.";

  while (!v.empty()) {
    int bits;
    auto f = v.front();

    if (f < 256) {
      bits = 8;
      v.erase(v.begin());
      return {f, bits};
    }
    if (f < 512) {
      bits = 9;
      v.erase(v.begin());
      return {f, bits};
    } else if (f < 1'024) {
    bits = 10;
    v.erase(v.begin());

    return {f, bits};
  } else if (f < 2'048) {
      bits = 11;
      v.erase(v.begin());

      return {f, bits};
    } else if (f < 4'096) {
    bits = 12;
    v.erase(v.begin());

    return {f, bits};
  } else if (f < 8'192) {
      bits = 13;
      v.erase(v.begin());

      return {f, bits};
    } else if (f < 16'384) {
    bits = 14;
    v.erase(v.begin());

    return {f, bits};
  } else if (f < 32'768) {
      bits = 15;
      v.erase(v.begin());

      return {f, bits};
    } else if (f < 65'536) {
    bits = 16;
    v.erase(v.begin());

    return {f, bits};
  } else
    throw "ERROR: Cannot create a bit length longer than 16 bits.";
 }
}

std::string int_to_binary_string(int value, int bits) {
 std::string s;
 // std::bitset<bits> b(value);
 // s.append(std::to_string(b));
```

```
 auto code = value;
 while (value > 0) {
  if (value % 2 == 0)
   s = "0" + s;
  else
   s = "1" + s;
  value = value >> 1;
 }
 auto zeros = bits − s.size();
 if (zeros < 0) {
  std::cout << "\nWarning: Overflow. code " << code
            << " is too big to be coded by " << bits << " bits!\n";
  s.substr(s.size() − bits);
 } else {
  for (auto i = 0; i < zeros;
       i++) // pad 0s to left of the binary code if needed
   s = "0" + s;
 }
 return s;
}

auto binary_string_to_int(std::string s) {

 auto code = 0;
 if (s.size() > 0) {
  if (s.at(0) == '1')
   code = 1;
  s.substr(1);
  while (s.size() > 0) {
   code = code << 1;
   if (s.at(0) == '1')
    code++;
   s.substr(1);
  }
 }
 return code;
}
```

```
#include "lzw435.hpp"
#include <cassert>
#include <fstream>
#include <iostream>
#include <iterator>

int main(int argc, char *argv[]) {
  // Throw an error if 3 parameters were not passed
  if (argc < 2) {
    std::cerr << "ERROR: no input files given\n";
    return 1;
  }

  std::string filename = argv[2];
  std::ifstream infile(filename.c_str(), std::ios::binary);

  // Throw an error if the file cannot be opened
  if (!infile.is_open()) {
    std::cerr << "ERROR: file cannot be opened. Check that the file exists.\n";
    return 2;
  }

  // Create a string of the file contents
  std::string in{std::istreambuf_iterator<char>(infile),
                 std::istreambuf_iterator<char>()};

  infile.close();

  try {

    switch (*argv[1]) {
    // Compress input file
    // If program was run passing c and filename to compress
    case 'c': {
      // Pass input file contents string to get vector of integers
      // representing integer value that corresponds to string in dictionary
      std::vector<int> v = compress(in);
      // Convert all integers to corresponding binary values
      // each appended to a string representing the file as a
      // string of binary numbers
      std::string output = int_to_binary_string(v, "");

      // Make the output string divisible by 8
      auto bit_length = output.length() % 8;
      output.append(bit_length, '0');

      assert(output.length() == (output.length() % 8 + output.length()));
      output = convert_to_bytes(output);

      filename.append(".lzw"); // Add .lzw extension to input file name

      std::ofstream out(filename.c_str(), std::ios::binary);
      out << output; // Output the compressed file contents to a new file

      out.close();
      break;
    }
    // Expand input file
    case 'e': {
      // filename2 should be identical to filename
      // separate the input into strings of length 12,
      // put these strings into a vector of integers
```

```
      in = make_string(in);
      std::vector<int> v = separate(in, 12);
      std::string d = decompress(v);
      filename.append("2"); // Save expanded file as filename2

      std::ofstream out(filename.c_str(), std::ios::binary);
      out << d;
      out.close();
      break;
    }
    }
  } catch (char const *err) {
    std::cout << "The library threw an exception:\n" << err << "\n";
  }

  return 0;
}
```