// Project 5 solution #include <iostream> #include <cstring> #include <cctype> using namespace std; const int MAX\_MESSAGES = 50; const int MAX\_MESSAGE\_LENGTH = 80; const int MAX\_NORMALIZED\_LENGTH = MAX\_MESSAGE\_LENGTH + 1; // +1 for blank after last word const int MAX\_ALL\_NORMALIZED\_LENGTH = MAX\_MESSAGES \* (MAX\_NORMALIZED\_LENGTH + 1); // +1 for newlines const int ALPHABET\_SIZE = 'z' - 'a' + 1; const char UNKNOWN = '?'; // any non-letter will do bool normalizeText(const char src[], char dest[], int destSize, bool preserveNewlines); bool matchCrib(const char text[], const char crib[], char plaintextLetters[]); bool matchCribStartingHere(const char text[], int textpos, const char crib[], char plaintextLetters[]); //////////////////// // decrypt // If originalCrib matches any message in ciphertext, write the (partially) // decrypted message and return true; otherwise, write nothing and return false. bool decrypt(const char ciphertext[], const char originalCrib[]) { // Normalize crib. If resulting crib would be too long to match any // message, or has no words, return false. (Length of array accounts // for one space after each word, including the last, and the '\0'.) char crib[MAX\_NORMALIZED\_LENGTH+1]; if (! normalizeText(originalCrib, crib, MAX\_NORMALIZED\_LENGTH+1, false)) // Don't preserve newlines return false; if (crib[0] == '\0') return false; // Normalize ciphertext. (Length of array accounts for one space after // each word, including the last, one newline after each message, // including the last, and the '\0'.) char normalizedCiphertext[MAX\_ALL\_NORMALIZED\_LENGTH+1]; if (! normalizeText(ciphertext, normalizedCiphertext, MAX\_ALL\_NORMALIZED\_LENGTH+1, true)) // Preserve newlines { cerr << "The total length of all messages exceeds the spec's guarantee!" << endl; return false; } // For a matching crib, this array will eventually contain the // decryptions of ciphertext letters. For example, if ciphertext // 'e' maps to plaintext 't', then plaintextLetters['e'-'a'] == 't'. // If a mapping is not determined, the element value is UNKNOWN. char plaintextLetters[ALPHABET\_SIZE]; // Try to match crib to a message. if (! matchCrib(normalizedCiphertext, crib, plaintextLetters)) return false; // If success, write decrypted text. for (int k = 0; k < ALPHABET\_SIZE; k++) plaintextLetters[k] = toupper(plaintextLetters[k]); for (int k = 0; ciphertext[k] != '\0'; k++) { char ch = ciphertext[k]; if (isalpha(ch)) { ch = tolower(ch); if (plaintextLetters[ch-'a'] != UNKNOWN) ch = plaintextLetters[ch-'a']; } cout << ch; } return true; } //////////////////// // normalizeText // Transform src string into dest string. If src has no words, dest will be // the empty string. Otherwise dest will contain all the words in src (all // lower cased), with one space character following each word (including the // last). Return true if this fits in dest (as determined by destsize, which // includes accounting for the trailing '\0'). Otherwise, return false (and // dest may be changed). // // As an example, " aBc--dE ?? fg#@hi" normalizes to "abc de fg hi " bool normalizeText(const char src[], char dest[], int destSize, bool preserveNewlines) { // Fill the dest string int d = 0; for (int s = 0; src[s] != '\0'; s++) { if (isalpha(src[s])) { // Transform each letter if (d >= destSize) return false; dest[d] = tolower(src[s]); d++; // Only the first non-letter after a word is mapped to a space if (!isalpha(src[s+1])) { if (d >= destSize) return false; dest[d] = ' '; d++; } } else if (preserveNewlines && src[s] == '\n') { if (d >= destSize) return false; dest[d] = '\n'; d++; } } if (d >= destSize) return false; dest[d] = '\0'; return true; } //////////////////// // matchCrib // If crib matches somewhere in the text, set plaintextLetters to // map ciphertext letters to plaintext letters and return true. Otherwise, // return false (and plaintextLetters might be changed to something useless). // The text and the crib must be normalized. bool matchCrib(const char text[], const char crib[], char plaintextLetters[]) { int k = 0; while (text[k] != '\0') { if (text[k] != '\n') { // Try matching starting at the word that starts at position k if (matchCribStartingHere(text, k, crib, plaintextLetters)) return true; // Skip to end of word while (isalpha(text[k])) k++; } // Skip the one space at the end of the word or skip a newline k++; } // No match anywhere return false; } //////////////////// // matchCribStartingHere // If crib matches the text starting at the word that starts at textpos, // set plaintextLetters to map ciphertext letters to plaintext letters and // return true. Otherwise, return false (and plaintextLetters might be // changed to something useless). The text and the crib must be normalized. bool matchCribStartingHere(const char text[], int textpos, const char crib[], char plaintextLetters[]) { // This will record whether we encounter in the crib a repeat of a // plaintext letter that appears earlier in the crib bool cribHasRepeatOfPrior[ALPHABET\_SIZE]; // Initialize: All mapping unknown, no repeat letters encountered yet for (int m = 0; m < ALPHABET\_SIZE; m++) { plaintextLetters[m] = UNKNOWN; cribHasRepeatOfPrior[m] = false; } // Walk through corresponding positions of crib and text. Since // both are normalized, a matching crib will have spaces in positions // that correspond to spaces in the text. The crib can't match across // a message boundary, because it can't match the newline. for (int c = 0; crib[c] != '\0'; c++) { char cribch = crib[c]; // crib character char textch = text[textpos+c]; // corresponding text character // If either is a space, the other must be, otherwise no match if (cribch == ' ') { if (textch != ' ') return false; } else if (textch == ' ') return false; // A crib letter can't match a newline in the text else if (textch == '\n') return false; // If neither is a space and we haven't matched this ciphertext // letter before else if (plaintextLetters[textch-'a'] == UNKNOWN) { // No other ciphertext letter can already map to this plaintext // letter. if (cribHasRepeatOfPrior[cribch-'a']) return false; // Record mapping plaintextLetters[textch-'a'] = cribch; cribHasRepeatOfPrior[cribch-'a'] = true; } // If we have matched this ciphertext letter before, it must map // to this plaintext letter else if (plaintextLetters[textch-'a'] != cribch) return false; } // We got through the crib without a mismatch return true; }