1. The translator class leaked memory when I tested it on g32. Unfortunately, I could not figure out where the memory was leaking, and I did not have time to reimplement my translator class in a different way. Apart from this, there are no known bugs.
2. MyHash class

My MyHash class uses a dynamic array of Bucket structs. Each bucket contains a pointer to itself, a pointer to the next bucket(this points to the next bucket within one array index), a Boolean to show whether the bucket is occupied, and stores one KeyType and one ValueType.

The reset function goes to each location in the array, and deletes any buckets that have been linked to the array bucket. It then deletes the main array of buckets and reassigns the array data member to a new array of 100 buckets.

The associate function first calculates a hash value for the key to be added. If the array element at the hash value position is empty, the bucket’s key and value are set to the function parameters, and occupied to true. If the array position at the hash value is occupied, the linked list at this position is traversed until an unoccupied bucket is reached, where the key, value and occupied status are set. If this added element causes the load factor to go above the maximum, a new array with double the buckets is created. For each index of the array, the bucket linked list is traversed, with the key and value in each bucket extracted and rehashed, and inserted in the correct index of the new array. If the bucket at the index is occupied, it traverses the linked list until it finds an empty spot. The old array is then deleted, the array data member set to the new array, and the data member holding total buckets is doubled.

The find function first finds the hash value for the key parameter, and goes to that index of the array. If that index is occupied and the key is the same as the key parameter, then a pointer to the value in the bucket is returned. Otherwise, the list is traversed till the end(nullptr) and if at some point the key is equal to the key parameter, the pointer of the value is returned. If an empty bucket is encountered, nullptr is returned.

Tokenizer class

For my tokenizer class, I used a vector of characters to hold the separators.

The tokenize function sets a variable startString to 0, then loops through each character of the parameter word. Within each loop, the function loops through the list of separators, checking whether a character of the input string is a token. If it is, a substring between startString and the current position-1 is added to the vector of strings to be returned, and startString is set to current position + 1. If all the characters have been looped through and startString is less that the length of the word, another substring is created from startString to the end of the string and added to the vector of strings, which is returned.

Word List class

For my word list class, I used a MyHash data member that maps strings to a vector of strings. The MyHash data member stores all letter patterns, and a list of words that follows a letter pattern.

I created a helper function called findStringPattern(string s) that returns the string pattern of a string s. It returns it in the form ABCCDE etc. where repeat letters are the same letter in the input string.

The function loadWordList creates an infile object to load a text file and loops the following until the end of the file. Within each loop, if a string is a valid word (no characters other than letters and apostrophes), it is converted to lower case, otherwise the loop proceeds to the next iteration. The word’s string pattern is then calculated, and the MyHash data member is checked to see if the pattern is already in the map. If it is, then the vector of strings that the pattern maps to is retrieved, the word added, and the map updated. If it is not already in the map, a new vector is created with only the current word in it, and the new pattern-vector association added to the hash table. Returns false if file cannot be loaded, true otherwise.

The contains function first converts the string parameter to lowercase. It then calls MyHash.find() on the string pattern of the string parameter. If it returns nullptr, the function returns false. Otherwise, the vector returned by the find function is searched for the parameter word. If it is found, it returns true, false otherwise.

The findCandidates function first calculates the string pattern of the cipher word and tries to find it in the hash table. If the vector returned is empty, the function returns this empty vector. Otherwise, for each word in the vector returned by find, each character is compared to equivalent character in currTranslation. If all non-question mark characters are equal, the word is added to a vector to be returned by the function. This is repeated for all words with the string pattern, with all words that satisfy the requirements added to the vector, which is returned at the end of the function.

Decrypter class

The decrypter class has a wordlist data member, a translator data member, a tokenizer data member and data member that is a vector of strings that will be returned by the function crack.

The constructor dynamically initializes the wordlist, translator and tokenizer, and the destructor deletes them.

The load function calls loadWordList on its wordlist data member.

The crack function operates by the algorithm provided in the specification. It runs step 1, then calls the helper function subcrack. Subcrack runs steps 2-7, and does not designate any further functionality to helper functions.

Translator class

My translator class uses a struct I created called stackElement. Each stackElement contains a hash table(MyHash class type) and a pointer to the stackElement previous in the stack(prev). The translator contains one data member, a pointer to the top of the stack.

The constructor creates a new StackElement, and inputs each character to the hash table as a key, all mapping to ‘?’. The destructor deletes each stackElement in the stack.

pushMapping first checks if both the parameter strings are valid(same length, only letters). If not valid, false is returned. Otherwise, the current mapping table is checked to see if it is compatible with the new mappings. For each letter of the plaintext, the program loops through to make sure there are no inconsistencies. If there is, false is returned. A new stack element is created, and the associations from the previous stack element are loaded into it. The function then loops through each letter of ciphertext/plaintext, and adds the new associations. This stack element is then set to the top of the stack, with the previous top of the stack accessible by the prev pointer.

popMapping deletes the stack element on top of the stack. If the top element is the only element of the stack, popMapping returns false, otherwise true.

getTranslation loops through each letter of the parameter string. For each letter, the corresponding value is looked up in the mapping on top of the stack, and this value is added to the return string. This return string is returned after all the characters corresponding values have been added to it.

1. My pushMapping method in the Translator class runs in O(N+NL) time rather than O(N+L), however both of these simplify to O(1), so the time complexity is satisfied if the word length as a maximum value.

All other methods in the other classes satisfy their Big-O requirements.