Project 4 specifications

September 2024

1 Overview and purpose

In this project, you will implement a number of string algorithms. Specifically, you will have an array of strings which you will sort, check for duplicates, search, and then search to find similar words such is used when suggesting the correct spelling of an incorrectly spelled word.

In this project, you will be guaranteed that no word is longer than 20 letters in length, and that all words will be purely alphabetic, so no hyphens or apostrophes, and all characters will be lower case. You will not have to check if a letter is upper case or lower case when making comparisons; however, you should keep this in mind while you are programming: what changes might you make in order to have this project work with "water" coming before "waterloo", which should come before "waterproof". Also, how should the contraction "we're" appear relative to "were". However, while these issues may be relevant to a real-world application, they are not relevant to this project.

2 Project description

2.1 String length

The first string function, `std::size_t length( char const *a )` calculates the number of characters in the string (the length) by counting the number of characters before the first null character `'\0'` and returning that number.

2.2 Comparing two strings

The second string function, `int compare( char const *str1, char const *str2 )`, will compare two null-character-terminated character arrays (strings) and returns

1. a negative integer (most reasonably `-1`, but any negative number is fine) if string `str1` comes in the dictionary before string `str2`,
2. `0` if the strings are equal, and
3. a positive integer (most reasonably 1, but any positive number is fine) otherwise (meaning that `str2` comes before `str1` in the dictionary.

Recall that a null-character terminated string is an array of type `char` where the character after the last character in the string is `'\0'`. Given two strings, we loop through the characters, so we start with $k = 0$.

1. If the k^{th} character of both arguments are both `'\0'`, then they are the same word, for all previous letters were the same.

2. If the k^{th} character of one argument is `'\0'`, then that word must come before the other, as the other has more letters. For example, “cat” comes before “catastrophe” in the dictionary.

3. If the k^{th} character of `a` is less than the k^{th} character of `b`, then return a negative integer. For example, looking at the first character ($k = 0$), “cat” comes before “dog”, and looking at the fourth character ($k = 3$), “them” comes before “then”.

4. If the k^{th} character of `a` is greater than the k^{th} character of `b`, then return a positive integer. For example, looking at the first character ($k = 0$), “luck” comes after “good”, and looking at the seventh character ($k = 6$), “comparison” comes after “compare”.

5. Otherwise, the characters are equal and we increment the index k.

2.3 Assigning a string

The third string function `void assign( char *str1, char const *str2 )` performs the overwriting of the characters at `str1` with the characters of `str2;`, so that following this operation, both strings up to and including the null character are the same.

In this project, we guarantee that all strings are 20 characters or less, so that 21 characters are allocated for each word, so there is no issue with there being sufficient space; however, in real-world applications, you have to make sure that there is sufficient space in the destination character array to store all the characters in the source array.

2.4 The distance between strings

The fourth string function `unsigned int distance( char const *str1, char const *str2 )` calculates the edit distance between the two strings. That is, the minimum number of insertions of a character, changing a character, or deleting a character that are required to convert one string into the other. For example, the edit distance between `fiction` and `friction`, and `contraction` and `contraption` both have a distance equal to one, while the distance between `celebration` and `calibration` is two. The distance between “substitution” and “subtracting” is 6, as you must: remove the second “s”, remove the “o”, append a “g”, and then change the characters “itu” with “rac”. You might want to try to convince yourself that this cannot be done with fewer than six edits.

In some cases, you will find algorithms described online where you are not able to actually derive how the algorithm works. However, this is a recursive algorithm:

1. The first base case is that `distance( "", str ) == length( str )`, so for example, the distance between `""` and `"hello"` is five, because you must add the five characters `'h'`, `'e'`, `'l'`, `'l'` and `'o'` to change the empty string into the string `"hello"`.

2. Similarly, `distance( str, "" ) == length( str )`, for the same reason.

3. If the first character of the two strings are the same, then the distance between the two strings must be equal to the distance between all subsequent letters.
   So, `distance( str1, str2 ) == distance( str1 + 1, str2 + 1 )`. For example, the distance between `cache` and `catch` is the distance between `ache` and `atch`, which is equal to the distance between `che` and `tch`.

4. If the first two characters are different, then we must either delete one of the characters, or change one leading character to another. Thus, in this case, `distance( str1, str2 )` is one (the act of deleting one character or switching it) plus the minimum of `distance( str1 + 1, str2 + 1 )`, `distance( str1, str2 + 1 )` or `distance( str1 + 1, str2 )`. For this function, you can use the `std::min` function, which is defined in the `algorithm` library of the C++ standard library.

   If you have questions about what `str1 + 1` does, see the section on tricks and tips.

What is not obvious, perhaps, is that this does indeed give you the edit distance between the two strings, but that is a topic for a more advanced course in algorithms. You should compare and contrast this only slightly more complex recursive algorithm as compared to the recursive definition of the binomial coefficients:

$$\text{binomial}(m,n) = \begin{cases} 1 & n = 0 \\ 1 & n = m \\ \text{binomial}(m, n-1) + \text{binomial}(m-1, n-1) & \text{otherwise.} \end{cases}$$

Aside: The distance between `form` and `from` is, according to the definition, above, two, but in reality, swapping two letters should be considered a single edit. Could you, not for this project, extend the definition above so that swapping two letters is counted only as a single edit? Hint: you'll have to add one more case between Cases 3 and 4.

2.5 Is an array of strings sorted

The first array function `std::size_t is_sorted( char *array[], std::size_t capacity )` will return `capacity` if the strings in the array follow a dictionary ordering. The return value is determined as described in the course notes: it is the index of the first entry that is less than the previous entry, or equal to `capacity` if the array is sorted. You may assume that characters are all lower case characters.

You can use the corresponding function discussed in class, but you will have to modify it to use the `compare(...)` function defined above. Make sure you test `compare(...)` before you use it here, otherwise, you won't know if a error is the result of your implementation of `is_sorted(...)` or your implementation of `compare(...)`.

2.6 The insert function

The second array function `void insert( char *array[], std::size_t capacity )` assumes the first `capacity - 1` entries of the array are sorted and then inserts the entry at `capacity - 1` into the correct location in the array, moving any string that appears after it in the dictionary to the right by one location.

This function will be very similar to the `insert(...)` function taught in class, but where the `value` variable will now be a character array, and you will have to dynamically allocate sufficient memory to fit the string at the end of the array. You will then use `assign(...)` and `compare(...)` to determine where to insert the word temporarily assigned to `value`.

2.7 The insertion sort function

The third array function `void insertion_sort( char *array[], std::size_t capacity )` is implemented as we saw in class, where we call the `insert(...)` function with appropriate arguments for values of k going from 1 to the capacity of the array.

2.8 Removing duplicates function

The fourth array function `std::size_t remove_duplicates( char *array[], std::size_t capacity )` assumes that the array is sorted and removes duplicate entries by shifting all unique strings to the left filling in any gaps left by the duplicate entries. The value returned is the number of unique entries. You don't have to store extra entries at the end.

2.9 Finding a string

The fifth array function `std::size_t find( char *array[], std::size_t capacity, char const *str )` walks through the array. If it finds the string `str`, it returns the index where the string was found. If the string is not in the array, it returns the index of the first entry of the array that has the smallest distance (as defined by the above `distance(...)` function). If there are multiple entries, only the first index is returned.

2.10 Reading words from a file

The function

```
void read_words_from_file(
  char const *filename,
  char **&word_array,
  std::size_t &num_words,
  std::size_t  max_length
);
```

reads from the file with the name specified in the first argument. The first line of the file is assumed to store a non-negative integer indicating how many words there are in the file, and this is assigned to `num_words`. Next, sufficient memory is allocated to store those words assuming that the maximum length is `max_length` (so we will have to allocate `max_length + 1` characters for each of the words. This will form an array of arrays, where the entry `word_array[k]` is a character array of `max_length + 1` entries.

Code for this function is provided in `main.cpp` (downloadable from Learn). You will need to look at how memory is allocated by this code in order to properly deallocate memory.

2.11 Deallocating memory

The function `void free_word_array( char** word_array )` deallocates the memory allocated for the array of character arrays by the `read_words_from_file(...)` function by calling `delete[] word_array[0]` and then `delete[] word_array`.

You will notice that there are only two calls to `new[...]` in `read_words_from_file(...)`, so we only require two calls to `delete[]` in this function.

2.12 Testing

The `int main()` provided in `main.cpp` may be modified to test your code. There is an example of how to declare an empty word array, read in words from the file `"test_words.txt"` and call various functions against the word array. There are a few words lists available for download from Learn but you can feel free to also make your own.

You are only required to submit project_4.cpp as the other files will be replaced with testing versions. You can add additional helper functions to your project_4.cpp file if you would like. Avoid having either a main() or read_words_from_file() function in your submission.

2.13 Compiling

Your first priority should be to create a code skeleton that compiles: Into the same directory, download:

```
p_4_header.hpp
main.cpp
```

You should then create a new project_4.cpp file. At the top of the file add: #include "p_4_header.hpp" From here your should create a code skeleton - an empty definitions of all the functions declared in the header file. Each function should either return zero (appropriate to the function's return type) or nothing if the function is void. For example:

```
#include "p_4_header.hpp"

std::size_t length( char const *a ) {
    return 0;
}

int compare( char const *a ) {
    return 0;
}

void assign( char *str1, char const *str2 ) {
}

...

void free_word_array( char** word_array ) {
}
```

Now that all the functions have a definition (although empty) you can compile your code. Use the command:

```
g++ -std=c++11 project_4.cpp main.cpp -o p4.exe
```

to compile your project using the main and header files provided. By doing this first you can keep compiling and testing as you complete each function.

3 Tricks and tips

Recall that the value of an array is an address, and therefore any address could be interpreted as the start of some array, whether or not the memory has been appropriately allocated. If the memory for a character array has been appropriately allocated so that there is at least $n + 1$ bytes allocated for a string of n characters, then observe that char *str is the local variable storing the address of the first character in the string, in which case str + 1 is the address of the second character, and so if we define char *str2{ str + 1 }, then the second string is all characters in the first string starting at str1[1]. Of course, if you make a change to str1 after the first character, this, too, changes the second string. For example, the following prints “brake” and “rake” and then “brace” and “race”:

```
#include <iostream>

// Function declarations
int main();

// Function definitions
int main() {
  char *str1{ "brake" };
  std::cout << str1 << std::endl;
  char *str2{ str1 + 1 };
  std::cout << str2 << std::endl;

  // Change the 'k' to 'c'
  str1[3] = 'z';

  std::cout << str1 << std::endl;
  std::cout << str2 << std::endl;

  return 0;
}
```