1 Project 1

Due: Feb 21 by 11:59p

Important Reminder: As per the course *Academic Honesty Statement*, cheating of any kind will minimally result in your letter grade for the entire course being reduced by one level.

This document first provides the aims of this project. It then lists the requirements as explicitly as possible. This is followed by a log which should help you understand the requirements. Finally, it provides some hints as to how those requirements can be met.

1.1 Aims

The aims of this project are as follows:

- To give you some exposure to programming in C++.
- To get you to write a `Makefile` from scratch.
- To allow you to familiarize yourself with the programming environment you will be using in this course.
- To give you exposure to the C++ standard library.

1.2 Requirements

You must push a `submit/prj1-sol` directory to your `master` branch in your github repository such that typing `make` within that directory will build a `wordcounts` executable with usage:

```
./wordcounts MAX_N_OUT MIN_WORD_LEN MAX_WORD_LEN FILE...
```

The program should output the most commonly occurring words in one-or-more files `FILE...`. The arguments are:

`MAX_N_OUT` A positive integer specifying the maximum number of words to be output.

`MIN_WORD_LEN` A positive integer specifying the minimum length of words to be output.

`MAX_WORD_LEN` A positive integer specifying the maximum length of words to be output. This must be greater than or equal to `MIN_WORD_LEN`.

`FILE...` The names of one-or-more files which will be the source of the words.

A **word** is a maximal sequence of contiguous non-whitespace characters to which the following sequence of transformations have been applied:

1. All non alphabetic characters are removed.
2. All upper-case characters are replaced by their lower-case equivalents.

For example, both the words `Didn't` and `didn't` are both transformed to `didnt`.

The output of the program should consists of up to `MAX_N_OUT` lines where each line contains:

1. A transformed word *W*.
2. A `:` followed by a single space character.
3. A count of the total number of words in the files `FILE...` which transform to word *W*.

The lines should be ordered in non-ascending order by the count. If two lines have the same count, then they should be ordered in descending order by the word (i.e. in dictionary order).

Note that the `MIN_LENGTH` and `MAX_LENGTH` command-line arguments constrain the length of the words after the transformations.

The program is also subject to the following non-functional requirements:

- It should detect all errors and report them to standard error with informative messages. The program may terminate after reporting its first error.
- It should be designed such that its running time should usually increase only linearly with the size of the files `FILE...`.
- It may not use any libraries other than those available with a standard C++ 17 installation.
- The program should be written to use C++ (rather than C) library functions as far as possible.
- The program should compile without any warnings.

1.3 Sample Log

An edited sample log of the operation of the program is shown below:

```
$ ./wordcounts
usage: ./wordcounts MAX_N_OUT MIN_WORD_LEN MAX_WORD_LEN FILE...
$ ./wordcounts 2 3 4x ~/cs240/labs/lab0/lab0.umt
bad integer value "4x" for MAX_WORD_LEN
$ ./wordcounts 2 4 3 ~/cs240/labs/lab0/lab0.umt
MIN_WORD_LEN 4 is greater than MAX_WORD_LEN 3
$ ./wordcounts 10 5 5 ~/cs240/labs/lab0/lab0.umt
```

```
using: 14
shell: 10
which: 10
clone: 8
files: 8
start: 8
email: 6
login: 6
where: 6
above: 5
$ ./wordcounts 10 5 6 ~/cs240/labs/lab0/lab0.umt
github: 34
should: 19
using: 14
branch: 13
course: 13
create: 13
shell: 10
which: 10
within: 9
access: 8
$ ./wordcounts 10 5 6 ~/cs240/labs/lab0/lab0.um
cannot read /home/umrigar/cs240/labs/lab0/lab0.um:
  No such file or directory
$
```

1.4 Hints

Three datatypes provided by the C++ library may prove useful:

`std:string` C's `char *` strings are extremely limited. `std::string` from the C++ library provides string which can grow dynamically (using `push_back()`) as well as supporting concatenation using a overloaded `+` operator. Additionally, they can be constructed easily from `char *` literals explicitly as in `std::string("hello")` or implicitly when a `char *` is used in a context where a `std::string` is expected. Hence it is possible to build up a dynamic message using something like `std::string("file \"") + fileName + "\" not found"`.

`std:vector<T>` C does not support arrays whose size can change dynamically. C++'s `std::vector` meets that need and allows storing sequences of any type `T` while provided array-like behavior using an overloaded `[]` operator.

`std:unordered_map<K, V>` This implements a mapping from keys having type `K` to values having type `V`. For example, it can maintain a map-

ping from words to their number of occurrences. Again, the `[]` operator is overloaded to allow code like `map[word] += 1`.

All three of the above datatypes manage their own memory.

The map and vector collections can be created by providing iterators over other collections specified using a inclusive starting "pointer" (often `begin()`) and exclusive ending "pointer" (often `end()`). For example, a vector can be created from `main()`'s `argv[]` array using:

```
std::vector<std::string>(argv, &argv[argc])
```

and from a map using:

```
std::vector<std::Pair<...>>(map.begin(), map.end())
```

where the type of the `Pair` depends on the type of the `map`.

The program can be organized as follows:

1. Convert appropriate command-line arguments from strings to integers, terminating with an error message if they are in error.
2. For each file name specified on the command line read in the contained words. As each word is read in, transform it and enter the transformed word into a map so as to track its count.
3. Once all the words have been read in, extract all the word-count pairs from the map into a vector. Sort the vector using the required ordering. Report up to `MAX_OUT_N` word-count pairs.

The following steps give a rough idea of how you can develop your program. It provides fragments of code but you will need to search the web to put everything together.

1. Use the procedure covered in *Lab 0* to create a new `prj1-sol` branch in your `i240?` github clone and set up a new `submit/prj1-sol` directory.
2. Use what you learnt in *Lab 1* to write a `Makefile` to build your program. Depending on how you choose to organize your program, it may be quite trivial. You should use this `Makefile` to recompile and test your code after each step below.
3. Start by writing a `main()` function. It takes 2 arguments `int argc` and a `char *argv[]` array referencing the actual `argc` command-line arguments to the program (the first argument corresponds to the program invocation).

You can convert from a C-style array of `char *` to a C++ `std::vector<std::string>` using an iterator over the `argv[]` array:

```
auto args = std::vector<std::string>(&argv[0], &argv[argc]);
```

4. Verify that the `MAX_N_OUT`, `MIN_WORD_LEN`, `MAX_WORD_LEN` are correct. You can convert from a `std::string arg` to an integer using `std::stoi(arg, endIndex)` and verify that `endIndex == arg.length()` to ensure that `arg` is an integer.
5. Write yourself a little test program to output the words from a file:
   (a) Create a `std::ifstream in(fileName)`. Then
   (b) While the state of the input stream `in` is good (using `in.good()`), you can read the next non-blank word into a `std::string` using `>>`.
   (c) When the state of `in` is no longer good, verify that it is indeed at end-of-file. If not, report an error.

   Something along the lines of:

```
std::ifstream in(fileName);
while (in.good()) {
  std::string w;
  in >> w;
  std::cout << w << std::endl;
}
if (!in.eof()) {
  //report error
}
```

   An alternative to the above which does not use the `good()` method:

```
std::ifstream in(fileName); //open file
if (!in) {
  //report error;
}
std::string w;
while (in >> w) {
  std::cout << w << std::endl;
}
if (!in.eof()) {
  //report error;
}
```

6. Incorporate the above into your program, but instead of outputting each word transform it as per the requirements. Then if the transformed word meets the length requirements, enter it in to a map.

```
std::unordered_map<std::string, Count> map;

...
Count& count = map[transformedWord];
count += 1;
```

   where `Count` is a suitable integral `typedef`.

   Once all the files have been read, the map will contain a count of all the words which meet the length requirements.

7. You need to sort the entries in the map, but a map is not directly amenable to sorting. Instead you need to extract the key-value pairs from the map into a `vector` using something like:

```
typedef std::pair<std::string, Count> WordCount;
auto wordCounts =
  std::vector<WordCount>(map.begin(), map.end());
```

   which can now be sorted using:

```
sort(wordCounts.begin(), wordCounts.end(),
     wordCountCompare)
```

   where `wordCountCompare()` is a suitable comparison function which returns **true** iff its two `WordCount` arguments meet the ordering specified for the project.

8. It is now a simple matter to output up to the first `MAX_N_OUT` pairs from these sorted `wordCounts`.

9. Iterate until you meet all requirements.

It is a good idea to commit and push your project periodically whenever you have made significant changes. When it is complete please follow the procedure covered in *Lab 0* to merge your `prj1-sol` branch into your `master` branch and submit your project to the TA via github.