NIU CSCI 340 GRADE-O-MATIC ASSIGNMENT

GENERIC ALGORITHMS AND HIGHER ORDER FUNCTIONS

INTRODUCTION

This assignment was designed to give students an opportunity to use some of the STL generic algorithms, and to write and use some higher order functions, composing them to make a more useful whole.

YOUR TASK

You are responsible for implementing, in `gen-algo.h`, all of the following functions:

- To load and process input data:
  - `size_t read_lines(std::istream &instream, FN linecb);`
  - `size_t split_string_strict(const std::string &instring, FN tokencb, char delim);`
  - `size_t split_string_greedy(const std::string &instring, FN tokencb, char delim);`
- To work with tables, which we will be using a vector containing other vectors as the rows to represent.
  - `void print_table(...);`
  - `int table_min_cols(const STR_TABLE &table);`
  - `std::vector <int> calc_widths(const STR_TABLE & table);`
  - `STR_TABLE load_tsv(const std::string & filename);`
  - `void table_sort_alpha(STR_TABLE & table, unsigned int col);`
  - `void table_sort_numer(STR_TABLE & table, unsigned int col);`

WHAT IS `STR_TABLE`?

The type you see above, `STR_TABLE`, is a `typedef` for `vector< vector<string> >`, which is being used as a table stored as a vector, containing vectors representing its rows, which contain string elements. Here are some examples of what to expect when working with it.

```
Given STR_TABLE x; :
  x[0]         => vector containing all elements in 0th row
  x[i]         => vector containing all elements in ith row
  (x[i])[j]    => jth column in ith row.
  x.size()     => number of rows in the table
  x[i].size()  => number of cells in the ith row
  x.begin()    => on the outer vector gives an iterator that points to the ***vector*** for
                  the 0th row
  x[i].begin() => (begin for inner vector) gives an iterator that points to the 0th element
                  of the vector for row i
```

FUNCTIONS FOR LOADING / PARSING DATA

These functions are higher order functions, which means that one or more of their parameters will be something that is callable as a function.

`read_lines( instream, linecb )`

- `instream` - a `std::istream` that you will read the lines from. This could be `cin` or an open file.
- `linecb` - a *callable* – a function or a function-like object that you can call with the `()` operator. For `read_lines` to work, the person calling it must give you a `linecb` that accepts a `std::string` as its only parameter.

This function will use `getline` on the `istream` passed in as `instream`. Every line in the file should be read, until the EOF. For each line read, call the “callback” function, linecb, and pass the string read to that function.

Your function must return the number of lines read.

`split_string_strict (instring, tokencb, delim )`

- `instring` - The string to be split.
- `tokencb` - something that can be invoked as a function, which takes a single string as a parameter. This function will be called once for each token detected.

- `delim` - The character used as the delimiter for splitting.

This function goes through the whole string passed in, splitting it into tokens based on the delimiter character. This is the **strict** version, which means that a delimiter always indicates the end of one token and the start of another, even if that means the token is empty.

When a token is detected, `tokencb` function must be called with a string containing that token.

The function will return the total number of tokens detected.

You may adapt your code from 02a for this purpose, or you can use this as an opportunity to do it in terms of the STL `find` and `find_if` algorithms.

`split_string_greedy (instring, tokencb, delim )`

- `instring` - The string to be split.
- `tokencb` - something that can be invoked as a function, which takes a single string as a parameter. This function will be called once for each token detected.
- `delim` - The character used as the delimiter for splitting.

This function goes through the whole string passed in, splitting it into tokens based on the delimiter character. This is the **greedy** version, which means:

- There is no such thing as an empty token unless the whole string was empty.
- Any delimiters that occur before the first token should be ignored, not included in the token.
- When there are several delimiters in a row, the first one splits the token, and the rest are ignored until a non-delimiter character begins the next token.

When a token is detected, `tokencb` function must be called with a string containing that token.

The function will return the total number of tokens detected.

You may adapt your code from 02a for this purpose, or you can use this as an opportunity to do it in terms of the STL `find` and `find_if` algorithms.

Table Functions

`print_table ( ost, table, widths, maxcol, pre, sep, post, leftalign)` This function is responsible for printing out your table in a tidy format.

- `ost` - The `std::ostream` to print the output to. This will usually be `cout`, but could be any `ostream`
- `table` - The `STR_TABLE` containing the data you need to print
- `widths` - `widths[i]` will contain the column width for column `i`
- `maxcol` - maximum number of columns to print
- `pre` - prefix - string to print before each row
- `sep` - separator - string to print between columns
- `post` - string to print at the end of each row
- `leftalign` - `bool` - `true` makes columns left-aligned, `false` aligns them to the right.

The column widths must be set using `std::setw`, and their alignment can be handled with `std::left` and `std::right`, which are declared in `<iomanip>`.

Do not print more than `maxcol` columns. This is for a couple of reasons:

- It can be used as an upper bound on `widths` if you only have a few column widths (to prevent segfault)
- There may have been rows that had uneven widths; this was detected, and the minimum width was passed here.

You can make the assumption that either the rows are all of the same length, or that `maxcol` that is passed in will be the width of the shortest row or less. If you use this outside of the test programs, that would be a thing to keep in mind.

`table_min_cols ( table )` This function looks at each of the rows in the table provided, and returns the width of the shortest row present.

If the input data was in the proper format, this should be the same as the maximum number as well, but this function is used to find out how many columns exist in all of the rows.

When implementing this, avoid writing a loop yourself. You should use the STL `for_each` algorithm, or `transform` together with `min_element`.

`calc_widths ( table )` This function returns a vector of integers. Each integer in the vector will contain the width (in characters) of the longest string in the corresponding column of the `STR_TABLE` passed in, `table`.

When implementing this, avoid writing a loop yourself. You should use the STL `for_each` algorithm, or `transform` together with `min_element`.

`load_tsv ( filename )` This function is responsible for loading a table from the file given, which must be a tab-separated values file. This data will be returned as a `STR_TABLE`.

Tab separated values files are a way of storing a table in a text format. Rows are delimited by the newline character, so if you read the whole line you get the whole row. The fields in each column are strictly delimited by a tab character (hence the name).

You are expected to do this using the `read_lines` function together with the `split_string_strict` function.

Hint: Use lambdas with capture by reference.

`table_sort_alpha( table, col )` Using the version of the `std::sort` algorithm that allows you to pass a custom comparator, order the rows in the table so that column number `col` is sorted in ascending order, alphabetically. The reference output used a case sensitive comparator.

`table_sort_numer( table, col )` Using the version of the `std::sort` algorithm that allows you to pass a custom comparator, order the rows in the table so that column number `col` is sorted in ascending order, numerically. This order will be different than the alphabetical sort for numbers of differing lengths. To convert a string value to a `double`, you can use the function `strtod`.

NOTES

There is one file that you should do your work in, `gen-algo.h`. Do not submit any other files.

You should feel free to create whatever files you want to test things locally, including writing your own simple programs to test small parts on their own, but none of them should be submitted to the autograder. I actually *encourage* you to write unit test programs for yourself, but they should not be a part of your submission.

TESTING

There are a number of testing programs included. These will be used to evaluate the functionality of your implementations of the required functions. Typing `make` will attempt to compile them all, and will succeed to the degree it can with whatever you have implemented at that point.

The table below has a list of the tests available, and they are shown in order from least complex to most complex.

Order	Test	Purpose
1	`test-read-line`	Tests your `read-line` function in isolation.
2	`test-simple-strict`	Tests your `split_string_strict` function in isolation.
3	`test-simple-greedy`	Tests your `split_string_greedy` function in isolation.
4	`test-print-table`	Tests your `print-table` function in isolation.
5	`test-calc-widths`	Tests `calc_widths`, requires `print_table`.
6	`test-sorting`	Tests your sort functions, requires `calc_widths` and `print_table`.
7	`test-load-tsv`	Tests your `load_tsv` function. Requires almost everything to work.
8	`test-load-sort`	Tests your `load_tsv` function together with sorting. This uses almost everything.

The expected output is contained in the `*.refout` files in the `output/` directory.

HOW TO SUBMIT

Submit your working `gen-algo.h` to the autograder.

Grading Considerations

- Does it compile? Does it run? All of the tests should compile and run on turing/hopper with the `Makefile` provided, but points will be deducted for each test that will not compile and run on the autograder.
- Does the output match for all of the tests? The autograder will compare the output of your program with the output of the working, reference implementation.
- Did you indent your code?
  - Indentation aids in the readability of source code, and if you're not indenting your code blocks, the grader will legitimately dislike you for it. I'm authorizing them to mark you off if you subject them to reading that.
- Did you document your code?
  - You need a docbox at the top of every one of the files you're required to change including:
    - Your name
    - Your zid
    - Your course section
    - A description of what the program does
  - You should add a docbox for every function that you implement, explaining what it does and what each parameter is for.
  - Add other comments inside your code blocks describing what you're doing and why.
  - The use of `doxygen` style comments is encouraged, but not required.