NIU CSCI 340 GRADE-O-MATIC ASSIGNMENT

WORKING WITH STL ITERATORS - ALGORITHMS

INTRODUCTION

In this assignment, the student will write several algorithms that use iterators to perform some simple computing tasks. They will be implemented using template functions so they will be able to be applied with any type of iterator (forward or better).

I have provided a `Makefile`, which will be used by the autograder to compile. If you download these files to Turing/Hopper, you can compile everything by typing `make` at the command line and hitting enter. Source code for two driver programs is provided:

- `do_ints` - a program that uses the functions you write to tokenize some sequences of integers and calculate various statistics along the way. You can look at the code in `do_ints.cc`, but do not make changes to that file.
- `do_strings` - a program that uses the functions you write to split up some strings. You can see the code in `do_strings.cc`, but do not make changes.

Both of these programs are complete **except for** the functions that are declared (but not defined) in `assign2-algos.decl.h`. You are responsible for implementing all of them. Make sure you put those in the `assign2-algos.h` file and nowhere else.

NOTES

This assignment will be submitted via the autograder. You will implement the required functions in the file `assign2-algos.h`, and submit that file to the autograder.

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 your submission should only include the required code.

ITERATOR ALGORITHMS - STATISTICS AND OUTPUT

In `assign2-algos.h`, you will need to implement **all** of the templated functions that were declared in `assign2-algos.decl.h`.

I recommend starting with the algorithms detailed in this section, as they will be slightly easier than the other set, which is described in the next section.

`range_sum` Iterate over the range provided, add up the elements, and return the sum. This is required to work with iterator ranges containing any type that can be meaningfully added into a `double`.

`range_avg` Iterate over the range provided, add up and count the elements, then calculate and return the average (mean) as a `double`.

`range_minval` Iterate over the range provided, keeping track of the minimum value found. Return that minimum value. Calling this on an empty range is undefined behavior, so don't worry about handling it.

`range_maxval` Iterate over the range provided, keeping track of the maximum value found. Return that maximum value. Calling this on an empty range is undefined behavior, so don't worry about handling it.

`range_count` Iterate over the range provided, counting the number of elements that occur within the range. Return that count.

`print_range( ost, begin, end, pre, sep, post, width )`

- `ost` is the `std::ostream` that the output should be printed to. In general, people will want to use `cout`, but you should use the one passed in so that this function can write to other streams when needed.
- `begin` is an iterator to the beginning of the range to be printed
- `end` is the iterator just past the last valid element of the range to be printed
- `pre` is a string that gets printed one time before anything else
- `sep` (separator) is a string that gets printed between each element (but not before the first or after the last)
- `post` is a string that gets printed after all of the elements have been printed
- `width` is the width, in characters, of the column in which the elements' values are printed, or 0 (zero) if you do not want to use a fixed size. Set `std::setw` for details.

This function should iterate over the range given, and print out the elements with the formatting determined by the other parameters. I've called it several times with different parameters in both driver programs, so you can get an idea of what's expected from the reference output, if it's not already clear to you.

`histogram( begin, end, bin_counts, N, width )` A histogram is a tool used in statistics where data is subdivided into several sub-ranges, which we will refer to as bins, and the frequency of occurrence of values within each range is measured. This is typically displayed graphically on a bar graph, but we just care about the numbers for this function.

The basic description of what you need to do for this function is to iterate over every element in the range `[begin, end)`, find out which bin that element belongs in, and increment the count for the appropriate bin. Make sure the counts are already at zero before you begin.

The first bin starts at `0`, and ends right before the second bin starts at `width`. The third bin starts at `2*width`, etc.

You can assume that any sequence provided for this will contain types that can be ordered (it'd break if you couldn't), and which can be meaningfully divided by a `double`.

- `begin` - iterator at the beginning of the sequence we're computing a histogram for
- `end` - iterator one past the last element of the sequence we're computing a histogram for
- `bin_counts` - an object supports the subscript `[]` operator to access integers containing the bin counts. The type is templated, so this could easily be an array or a `vector`, but it could also be any object that can be used in the appropriate way. This function will assume that it is already set up so you can read from and write to integers with indices from `bin_counts[0]` to `bin_counts[N-1]` without a problem.
- `N` - the total number of bins, and an upper bound on the indices used with `nums`
- `width` - The width of each bin. You can divide an element's value by this and use the integer part of the quotient to decide which bin this fits in. If the quotient is less than zero or greater than `N-1`, then do not increment any of the bins.

ITERATOR ALGORITHMS FOR SPLITTING RANGES

These functions, which you will implement using the iterator pattern, are useful for splitting a long sequence into smaller sub-sequences, which we shall call tokens.

Although the `do_strings` driver program will use a `std::string` and the `do_ints` driver program will use a `std::vector` or a `std::list` to provide the iterators your functions will be called with, you should make sure that your algorithms are written in such a way that *any* forward iterator (or better) can be used without having to rewrite your function.

We will likely not yet have covered *exactly* what that means in lecture, but make sure you follow these rules:

- While you can store your iterator's current position into another iterator to keep track of where you were, you cannot rewind an iterator. Assume there is no `--` available for that.
- Do not assume that it's possible for your iterator to move more than one step at a time. Do not try to use pointer arithmetic or subscript operators, as this needs to be able to work on sequences that don't support random access.

Note that `std::string` can work like any other STL container, in that it has `begin()` and `end()` methods that can be used to iterate over the individual characters in the string. Similarly, you could use these algorithms with character arrays if you pass a pointer to the 0th element as its "begin", and a pointer to the element after the last valid character (where the null character would be in a null-terminated string) as its "end".

`token_info` This a `struct` (If you're not familiar with structs, think of them as classes but with everything public by default). It will be used by the `get_next_token_*` functions to track where successive calls left off so they can continue until the end.

- `t_begin` - Your functions will be responsible for setting this iterator to point to the beginning of the token found.
- `t_end` - Your functions will be responsible for setting this iterator to point right after the last element of the token found.
- `t_next` - Your functions will be responsible for setting this iterator to point the beginning of the next token, or to then actual end of the range if there are no further tokens.
- `begin()` - Returns an iterator to the beginning of the token that was found. This is useful because it allows you to use the range-based for loop on this to iterate over the sub-range represented.
- `end()` - Returns the iterator representing the end of the half-open range representing the token that was found.

`get_next_token_strict ( begin, end, delim )`, `get_next_token_greedy ( begin, end, delim )` These functions will start at `begin` and iterate until it either reaches the `end` iterator, or finds an element in the range that has the value `delim`.

Your function will be responsible for setting the members of the `token_info` returned appropriately. The `t_begin` member will be set to the iterator representing the beginning of the token, and the `t_end` member will be set to the location immediately past the end of the token, or to `end` if we reach the end of the input range.

The `t_next` member of the `token_info` struct returned must point to the beginning of the next token, or to the end of the range if no more tokens are to be found. How this is determined will be the main difference between the "strict" and "greedy" versions.

- `begin` - Where to begin the search for a delimiter.
- `end` - The end of the range to search. If we reach this, there are no more elements to form tokens with.
- `delim` - the delimiter value, when it is found in the range, it is used to indicate that one token has ended

Strict vs. Greedy For the "strict" version, every delimiter begins a new token, and a delimiter found immediately after another delimiter will indicate the presence of a zero-length token. This is useful for situations like tab- or comma-separated values files, where you want the delimiter to start a new field and would actually like to be able to leave fields blank.

For the "greedy" version, once a delimiter is found, the next token starts at the next position that is not a delimiter. That means that if we have several delimiters in a row, they are "eaten" and do not indicate empty tokens. One time this can be useful is in handling for whitespace in strings when you don't want the number of spaces to matter, which is the case when parsing HTML or similar languages.

```
// Let's do an example tokenizing of integers using 0 as the delimiter
   source sequence: { 100 200 300   0   0   0   5   6   0   7   8   9 }

   strict token 0: { 100 200 300 }
   strict token 1: { }    // empty
   strict token 2: { }    // empty
   strict token 3: {   5   6 }
   strict token 4: {   7    8   9 }

   greedy token 0: { 100 200 300 }
   greedy token 1: { 5 6 }
   greedy token 2: { 7 8 9 }
```

HOW TO SUBMIT

To submit this program, you will upload the required files to the NIU Autograder on Blackboard.

GRADING CONSIDERATIONS

- Does it compile? Does it run? If it doesn't compile and run on turing/hopper with the `Makefile` provided, no points will be awarded.
- Does the output match? The output from your implementation will be tested against the output from the reference implementation for both driver programs. I have provided reference output for each of the driver programs, so you can compare your program's output in advance. This output can be found in `do_ints.refout` and `do_strings.refout`.
- 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.