Linköping University
Department of Computer and Information Science (IDA)
The UPP group 2015-10-01

Standard Template Library

Aim

In this assignment you will learn how to use the standard template library.

Reading instructions

- String features (`std::string`, `at()`, `substr()`, `insert()`, `erase()`, `<cctype>`)
- STL containers (`std::vector`, `std::list`, `std::map`)
- STL iterators (`::iterator`, `::const_iterator`)
- STL algorithms (`sort()`, `find()`, `copy()`, `for_each()`)
- Passing functions as arguments
  - Function pointers (the C way)
  - Function objects (`::operator()`, the C++98 way)
  - Lambda functions (the C++11 way)
- Regular expressions, **voluntary, may require gcc version >= 4.9**, (`std::regex`, `std::regex_match`, `std::regex_replace`, `std::regex_search`)

Word list background

A few years back, before smartphones, mobile phones tried to predict what you were trying to write by using a word list. Sometimes successfully, and sometimes yielding a good story to tell your friends (after the initial awkwardness and apologizing to the recipient of your message). Does it sound familiar?

A less known fact is how the word list used by your phone was created. This story[1] may thus come as a revelation. It starts in a beautiful far away country. The labor is both good and cheap. An ideal place for a phone company to locate the word list department. In this department they developed a program to crawl the web and collect words and word statistics from all popular web pages. To determine the language of the words they simply looked at the top domain where a page occurred. With this collection of words and their usage frequency the word list department could compile word lists which were able to provide good (or at least funny) predictions.

A few native far-away-countrymen also did their best at excluding words that were not proper (unless they could make for funny misunderstandings of course).

[1] If there's any truth to the story it's purely coincidental.

Word collection and washing

Some emerging markets still use the older phones for their increased battery time and coverage. They need a new word collecting program. You will write it for them now. The program should scan a text file for *potential words*. All potential words are separated by at least one blank character. A potential word may happen to include some "junk" normally occurring in written text, for example opening and closing parentheses, citations, commas, dots and other characters that may end a sentence. We separate this in junk that may occur just before the actual word, and junk that may occur just after. Possible junk are specified as follow:

- Junk characters to remove from the beginning of *potential words* are quotation marks and opening parentheses. We call this leading junk for short: `"'(`
- Junk characters to remove from the end of *potential words* are punctuation characters, quotation characters, apostrophes and closing parentheses. We call this trailing junk for short: `!?;,:."')`
- Remaining after trailing junk removal may also occur one single `'s` (possession/genitive) that should be removed if present.
- Note that junk characters inside the word are left in place, since they're neither leading nor trailing.

Your program will strip away only leading and trailing junk from each *potential word*. This will produce a *cleaned word*. Cleaned words are then determined to be either *valid* or *invalid*. A *valid* word have the following properties:

- The word contains only letters (lower case or upper case) and hyphens.
- Hyphens occur only inside the word (not first or last) and non-consecutive.
- The word contains at least three characters (shorter words are not worthwhile to predict).

Once a word is determined as *valid*, it is converted to lower case and added to the word list and statistics. *Invalid* words are simply ignored.

Program input and output

A non-functional requirement (that's still a requirement) is to use STL containers and algorithms wherever possible (rather overdo it than miss out on a learning opportunity).

Your program should work on a plain text file (any text file). Interesting test cases include the source code of the program itself and any HTML file you can find. The file is specified by the user on the command line (check for errors) and read by the program according to the word collection and washing section. A second command line parameter (after the file name) specify the final output (see later examples):

-a All *valid* words are printed in alphabetic increasing order followed by the frequency of that word (how many times that word occurred). This list should be formatted clearly with the first letter of each word and the last digit of each number in straight columns. You have to adapt to the longest word and largest number.

-f All *valid* words are printed in decreasing frequency. The list is printed as before, but with the last letter of each word aligned in a straight column instead of the first (right alignment). You will probably have to copy the words to another container before sorting.

-o Print all *valid* words in the same order they appear in the original file.[2] Insert line breaks to keep all lines as long as possible, but strictly below N characters[3] excluding only the line break character. N is specified by the user last on the command line (just after `-o`). Specifically, this should be solved by using the `for_each` algorithm. You will need a function object or a lambda function (why not try both ways).

Next page provide a small example of how the program is intended. You will of course have to create more elaborate test cases yourself if none are given. It is your responsibility to prove that your program is correct.

[2] This is for the quality assurance team to easier read the file content of for example a HTML file.

[3] Words longer than N characters will be printed on its own line, ignoring the limit.

An example HTML file

```
<html>
<head>
<title> The page title! </title>
</head>
<body id="my-body"><h1>The Page: </h1><p>This is the page body. </p></body>
</html>
```

Running the program with the example file

```
$ a.out
Error: No arguments given.
Usage: a.out FILE [-a] [-f] [-o N]

$ ./a.out example.html
Error: Second argument missing or invalid.
Usage: ./a.out FILE [-a] [-f] [-o N]

$ a.out example.html -a
body   1
page   3
the    2
title  1

$ a.out example.html -f
 page  3
  the  2
 body  1
title  1

$ a.out example.html -o 14
the page
title page
the page body

$ a.out example.html -o 9
the page
title
page the
page
body
```