$Id: regex-match.mm,v 1.40 2022-02-07 22:07:37-08 - - $
/afs/cats.ucsc.edu/courses/cse111-wm/Assignments/asg3-listmap-templates/discussion-misc.d
https://www2.ucsc.edu/courses/cse111-wm/:/Assignments/asg3-listmap-templates/discussion-misc.d/

1. Regular expressions

Regular expressions are a powerful way of scanning lines of text and selecting parts thereof based on pattern specifictions.

In C++, the facility is made available via **<regex>**.

https://www.cplusplus.com/reference/regex/

2. https://regexr.com/

https://regexr.com/ has an interactive page to learn about regular expressions.

3. Raw strings

Raw strings are a way of writing string constants without the need to escape escapes. For example, the string **"\\\"\t\n"** represents a backslash, a quote, a tab, and a newline.

But regexes are represented as strings, and use backslashes for a separate semantic reason. To avoid having to double every backslash, a raw string can be used. This is denoted by placing the letter **R** in front of the quoted string, and using parentheses inside the string.

So, for example **R"(\"\t)"** represents the actual string of 4 characters that appears between the parentheses — a backslash, a quote, a backslash, and the letter "t".

4. Regex classes in C++

Consider the example program **matchlines.cpp**. It has the following three declarations :

```
regex comment_regex {R"(^\s*(#.*)?$)"};
regex key_value_regex {R"(^\s*(.*?)\s*=\s*(.*?)\s*$)"};
regex trimmed_regex {R"(^\s*([^=]+?)\s*$)"};
```

regex is the data type initialized with the raw strings. Pattern matching applies a regular expression to a string and determines a match. Regexes are a programming language in themselves.

Dissection of the strings. In the following each item shows only the actual regex after the raw string delimiters have been stripped.

(a) `^\s*(#.*)?$`
Match optional white space, optionally followed by a hash and anything that follows it.

- `^` Match the beginning of the string.
- `\s*` Match zero or more white space characters. Backslash escapes meta characters and makes ordinary characters have special meanings. `\s` match white space (spaces, tabs, newlines). The asterisk indicates that the preceding item should be recognized zero ore more times.
- `(` Begin a capture. In a **`regex_search`**, this will be an element of the result vector.
- `#.*` Match the hash (literally), followed by zero or more of anything. A dot (`.`) matches any characters except a newline, and (`.*`) matches zero or more of any character.
- `)?` End the capture and place it into the result. The question mark indicates that the preceding capture is optional.
- `$` Match the end of the string.

(b) `^\s*(.*?)\s*=\s*(.*?)\s*$`
Match any sequence of characters preceding an equal sign (`=`) and then also after the equal sign, capturing sequences befoe and after it. In each sequence white space is trimmed fore and aft.

- `^` Beginning of line.
- `\s*` Zero or more white space.
- `(.*?)` Match and capture any sequence of zero or more characters. Non-greedy (lazy) matching, as few as possible.
- `.*` would mean a greedy match, but
- `.*?` is a lazy match, which matches as few characters as possible.
- `\s*` Zero or more white space.
- `=` Match an equal sign.
- `\s*` Zero or more white space.
- `(.*?)` As above, the second captured match. Parentheses indicate a capture. Non-greedy matching of zero or more characters.
- `\s*` Zero or more white space.
- `$` End of line.

(c) `^\s*([^=]+?)\s*$`
Match a line containing a sequence of one or more characters none of which is an equal sign.

`^` Beginnng of line.

`\s*` White space.

`([^=]+?)` Capture what is in the parentheses. Brackets indicate a set of characters, in this case the character equal sign. The circumflex (hat) complements the set. So

`[^=]` matches any character not an equal sign.

`[^=]+` matches one or more such characters.

`[^=]+?` matches one or more such characters, but in a non-greedy (lazy) manner, matching as few as possible.

`\s*` White space.

`$` End of line.

5. Usage of regex

The `regex` variables declared above are used in the example program `matchlines.cpp`. Part of the code follows.

(a) `string line;`
`smatch result;`
An `smatch` variable holds the result of a regex match, and is used to store the results of a search. Each pair of parentheses in the regex will capture a matched result.
- `result[0]` represents the entire matched expression
- `result[i]` represents sub-expression `i` that has been matched.

(b) `if (regex_search (line, result, comment_regex)) {`
Search the line to see if it matches the comment regex. If so, we can ignore the line because comments are not data. `regex_search` returns true if the match has succeeded.

(c) `}else if (regex_search (line, result, key_value_regex)) {`
Search the line for a key value pair. This regex has two captures. If the match succeeds, the `smatch result` variable has two values in it. At this point:
- `result[1]` has the key (first captured string)
- `result[2]` has the value (second captured string)

(d) `}else if (regex_search (line, result, trimmed_regex)) {`
Search the line for the trimmed regex (the third alternative). This is the query for the program.
- `result[1]` has the value of the query string that was captured.

(e) `}else { assert(false)` ...
This can't happen if the three regexes are exhaustive. But an `assert(false)` just does a backup check to make sure the program crashes if the logic is wrong.

6. Complete code for `misc/matchlines.cpp` example

```
#include <cassert>
#include <iostream>
#include <regex>
#include <string>
using namespace std;

int main() {
   regex comment_regex {R"(^\s*(#.*)?$)"};
   regex key_value_regex {R"(^\s*(.*?)\s*=\s*(.*?)\s*$)"};
   regex trimmed_regex {R"(^\s*([^=]+?)\s*$)"};
   for (;;) {
      string line;
      getline (cin, line);
      if (cin.eof()) break;
      cout << "input: \"" << line << "\"" << endl;
      smatch result;
      if (regex_search (line, result, comment_regex)) {
         cout << "Comment or empty line." << endl;
      }else if (regex_search (line, result, key_value_regex)) {
         cout << "key  : \"" << result[1] << "\"" << endl;
         cout << "value: \"" << result[2] << "\"" << endl;
      }else if (regex_search (line, result, trimmed_regex)) {
         cout << "query: \"" << result[1] << "\"" << endl;
      }else {
         assert (false and "This can not happen.");
      }
   }
   return 0;
}
```