14.6. Regex Flags

We already introduced some regex constants you can use to influence the behavior of the regex interfaces:

Click here to view code image

```
regex reg3("<\\(.*\\)>.*</\\1>",regex_constants::grep);     // use grep
grammar

regex_replace (data, reg,
               string("<\\1 value=\"\\2\"/>"),
               regex_constants::format_sed)      // use sed replacement syntax
```

But there is more. Table 14.2 lists all regex constants provided by the regex library and where they can be used. In principle, they can always be passed as the optional last argument to the regex constructor or to the regex functions.

Table 14.2. Regex Constants of Namespace `std::regex_constants`

regex_constants	Meaning
Regex Grammar:	
ECMAScript	Use ECMAScript grammar (default).
basic	Use the basic regular expression (BRE) grammar of POSIX.
extended	Use the extended regular expression (ERE) grammar of POSIX.
awk	Use the grammar of the UNIX tool awk.
grep	Use the grammar of the UNIX tool grep.
egrep	Use the grammar of the UNIX tool egrep.
Other Creation Flags:	
icase	Ignore case-sensitivity.
nosubs	Don't store subsequences in match results.
optimize	Optimize for matching speed rather than for regex creation speed.
collate	Character ranges of the form [a-b] shall be locale sensitive.
Algorithm Flags:	
match_not_null	An empty sequence shall not match.
match_not_bol	The first character shall not match the *beginning-of-line* (pattern ^).
match_not_eol	The last character shall not match the *end-of-line* (pattern $).
match_not_bow	The first character shall not match the *beginning-of-word* (pattern \b).
match_not_eow	The last character shall not match the *end-of-word* (pattern \b).
match_continuous	The expression shall match only a subsequence that begins with the first character.
match_any	If more than one match is possible, any match is acceptable.
match_prev_avail	The positions before the first character is a valid positions (ignores match_not_bol and match_not_bow).
Replacement Flags:	
format_default	Use default (ECMAScript) replacement syntax.
format_sed	Use replacement syntax of the UNIX tool sed.
format_first_only	Replace the first match only.
format_no_copy	Don't copy characters that don't match.

Here is a small program that demonstrates the usage of some flags:

Click here to view code image

```
// regex/regex4.cpp

#include <string>
#include <regex>
#include <iostream>
```

```
using namespace std;

int main()
{
    // case-insensitive find LaTeX index entries
    string pat1 = R"(\\.*index\{([^}]*)\})";        // first capture group
    string pat2 = R"(\\.*index\{(.*)\}\{(.*)\})";   // 2nd and 3rd capture
group
    regex pat (pat1+"\n"+pat2,
               regex_constants::egrep|regex_constants::icase);

    // initialize string with characters from standard input:
    string data((istreambuf_iterator<char>(cin)),
                istreambuf_iterator<char>());

    // search and print matching index entries:
    smatch m;
    auto pos = data.cbegin();
    auto end = data.cend();
    for ( ; regex_search (pos,end,m,pat); pos=m.suffix().first) {
        cout << "match: " << m.str() << endl;
        cout << "  val: " << m.str(1)+m.str(2) << endl;
        cout << "  see: " << m.str(3) << endl;
    }
}
```

The goal is to find LATEX index entries that might have one or two arguments. In addition, the entries might use lowercase or uppercase mode. So, we have to search for either of the following:

- A backlash followed by some characters and `index` (lower- or uppercase) and then the index entry surrounded by braces for something like the following:

Click here to view code image

```
\index{STL}%
\MAININDEX{standard template library}%
```

- A backlash followed by some characters and `index` (lower- or uppercase), and then the index entry and a “see also” entry surrounded by braces for something like the following:

```
\SEEINDEX{standard template library}{STL}%
```

Using the `egrep` grammar, we can put a newline character between these two regular expressions. (In fact, `grep` and `egrep` can search for multiple regular expressions at the same time, specified in separate lines.) However, we have to take *greediness* into account, which means that we have to ensure that the first regular expression does not also match the sequences that should match the second regular expression. So, instead of allowing any character inside the index entry, we have to ensure that no braces occur. As a result, we have the following regular expressions:

```
\\.*index\{([^}]*)\}
\\.*index\{(.*)\}\{(.*)\}
```

which can be specified as raw strings:

```
R"(\\.*index\{([^}]*)\})"
R"(\\.*index\{(.*)\}\{(.*)\})"
```

or as regular string literals:

```
"\\\\.*index\\{([^}]*)\\}"
"\\\\.*index\\{(.*)\\}\\{(.*)\\}"
```

We create the final regular expression by concatenating both expressions and passing the flags to use a grammar in which `\n` separates alternative patterns (see Section 14.9, page 739) and to ignore case sensitivity:

Click here to view code image

```
regex pat (pat1+"\n"+pat2,
           regex_constants::egrep|regex_constants::icase);
```

As input, we use all characters read from standard input. Here, we use a string `data`, which is initialized by begin and end of all characters read (see Section 7.1.2, page 256, and Section 15.13.2, page 830, for details):

Click here to view code image

```
string data((istreambuf_iterator<char>(cin)),
            istreambuf_iterator<char>());
```

Now note that the first regular expression has one capture group, whereas the second regular expression has two capture groups. Thus, if the first regex matches, we have the index value in the first subgroup. If the second regex matches, we have the index value in the second submatch and the "see also" value in the third submatch. For this reason, we output the contents of the first plus the contents of the second submatch (one has a value and the other is empty) as value found:

Click here to view code image

```
smatch m;
auto pos = data.begin();
auto end = data.end();
for ( ; regex_search (pos,end,m,pat); pos=m.suffix().first) {
    cout << "match: " << m.str() << endl;
    cout << "  val: " << m.str(1)+m.str(2) << endl;
    cout << "  see: " << m.str(3) << endl;
}
```

Note that calling `str(2)` and `str(3)` is valid even if no match exists. `str()` is guaranteed to yield an empty string in this case.

With the following input:

Click here to view code image

```
\chapter{The Standard Template Library}
\index{STL}%
\MAININDEX{standard template library}%
\SEEINDEX{standard template library}{STL}%
This is the basic chapter about the STL.
\section{STL Components}
\hauptindex{STL, introduction}%
The \stl{} is based on the cooperation of
...
```

the program has the following output:

Click here to view code image

```
match: \index{STL}
  val: STL
  see:
match: \MAININDEX{standard template library}
  val: standard template library
  see:
match: \SEEINDEX{standard template library}{STL}
  val: standard template library
  see: STL
match: \hauptindex{STL, introduction}
  val: STL, introduction
  see:
```