C++11 Regular-Expression Library

By Brian Overland

Jun 25, 2013

This chapter is from the book

C++ for the Impatient

Learn More Buy

20.3. Constructing a RegEx String

The previous two sections provided an introduction to regular-expression patterns. The next several sections summarize the syntax rules, beginning with the syntax for matching individual characters.

This chapter adopts the default grammar used by the C++11 regular-expression library, which is a modified ECMAScript grammar. Although it's possible to use variations, the C++11 default is more versatile and expressive than the alternative grammars.

20.3.1. Matching Characters

The following special expressions match an individual character belonging to a group, such as letters or digits. This section also describes special conditions such as beginning-of-line or word boundary.

In the following list, a *range* may be a list of characters (not separated by spaces or commas, which themselves are characters). A *range* may optionally use a dash (minus sign) to indicate a run beginning with one character, up to and including another. Characters are ordered according to their underlying numeric (ASCII) value. For example, "[a-z]" matches all lowercase letters.

▶ .

Matches any one character other than a newline. For example, the following pattern string matches almost any single character:

```
"."
```

▶ **[***range***]**

Matches any one character in the specified range. For example, the following pattern string

matches any single letter in the range “a” to “m”. It also matches “z”.

```
"[a-mz]"
```

Most characters lose their special meaning inside the brackets. The minus sign gains special meaning to indicate a run of characters as in the example just shown, but only if it appears between two other characters inside the range. The following expression matches any one of the characters “+”, “*”, “/”, or “-”. None of these need to be escaped.

```
"[+*/-]"
```

[^*range*]
Matches any character *not* in the specified range. For example, the following pattern string matches any single character *other* than “a”, “b”, or “c”:

```
"[^abc]"
```

\n
Matches a newline. When using this in a C++ literal string meant to be part of a regular-expression pattern-matching string (as opposed to an actual embedded newline), remember that two backslashes must be used. For example:

```
"\\n"
```

\t
Matches a tab character.

\f
Matches a form feed.

\r
Matches a carriage return.

\v
Matches a vertical tab.

\x*hh*
Matches a character specified as a hexadecimal code. For example:

```
"\\xf3"
```

\u*hhhh*
Matches a Unicode character specified as a hexadecimal code. For example:

```
"\\u02f3"
```

\d
Matches any digit character. This is equivalent to:

```
[0-9]
```

\D
Matches any character other than a digit. This is equivalent to:

```
[^0-9]
```

\s
Matches any whitespace character.

\S
Matches any character other than a whitespace character.

\w
Matches any digit, letter, or underscore.

\W
Matches any character other than a digit, letter, or underscore.

\b
Matches a word boundary. A word must begin or end at this position, or there is no match. Words are made up of alphanumeric characters and are delimited by whitespaces and punctuation. For example, the following string matches any word beginning with “c” and ending with “t”, such as “cat” or “containment”. It does not match “caution”.

```
"\\bc[a-zA-Z]*t\\b"
```

\B
Not a word boundary. For example, the following pattern matches a portion of a word beginning with “a.” It will match “at” embedded in “cat”, but it will not match “at” if it occurs as a stand-alone word.

```
"\\Ba[a-zA-Z]*"
```

^
Beginning of line: The next character is matched only if it is the first character in the text to be examined or occurs just after a newline.

```
"^Barney"
```

$
End of line: The previous character matched (if any) must be the last character in the line of text.

20.3.2. Pattern Modifiers

Regular-expression pattern matching becomes more interesting when you modify a pattern to indicate possible repetitions. This feature, as much as anything else, makes the regular-expression technology a powerful and versatile tool for searching and replacing text.

In the following list, *expr* is an expression. For example, in the string “[0-9]+”, “[0-9]” is an expression and the + operator modifies its meaning.

An operator associates with the character closest to it, except where brackets or parentheses are used, in which case the operator refers to the whole range or group that precedes it.

▶ *expr**

Matches zero or more instances of *expr*. For example, the following string matches an empty string or a digit string:

```
"[0-9]*"
```

▶ *expr+*

Matches one or more instances of *expr*. For example, the following string matches a digit string of length one or greater.

```
"[0-9]+"
```

▶ *expr?*

Matches either one or zero instances of *expr*. The *expr* thereby becomes an optional item that can appear at most once. For example, the following string matches a minus sign or an empty string.

```
"-?"
```

▶ *expr1*|*expr2*

Matches *expr1* or *expr2*, but not both. For example, the following regular-expression string matches “aa” or “bb” but not “aabb”.

```
"(aa)|(bb)"
```

This expression can be made optional by placing it in a larger group and then using the ? operator. In that case, “aa” may appear, “bb” may appear, or they may both be omitted.

```
"((aa)|(bb))?"
```

▶ *expr*{*n*}

Matches exactly *n* instances of *expr*. For example, the following pattern string matches a target string containing exactly ten copies of capital “A”.

```
"A{10}"
```

▶ *expr*{*n*,}

Matches *n* or more instances of *expr*. For example, the following pattern string matches a target string consisting of three or more digits.

```
"[0-9]{3,}"
```

▶ *expr*{*n*,*m*}

Matches at least *n*, but no more than *m*, instances of *expr*. For example, the following pattern string matches a digit string no more than seven digits long.

```
"[0-9]{1,7}"
```

(*expr*)
Forms a group. *expr* is considered as a unit when modified by other special characters, as in (*expr*)+, (*expr*)*, and so on. For example, the following pattern string matches "AbcAbcAbc" in the target string:

```
"(Abc){3}"
```

Another important effect of parentheses is that they cause the expression inside to be "tagged," as explained in the next section.

20.3.3. Recurring Groups

Much of the power of regular expressions comes from the ability to look for repetitions of a group. The syntax:

```
\n
```

refers to a previously tagged group. The expressions \1, \2, and \3 refer to the first three groups. Remember that C++ string literals use the backslash as an escape character, so the expressions "\1", "\2", and "\3" must be rendered as \\1, \\2, and \\3, and so on, in C++ source code (unless you're using raw string literals).

For example, the following expression—expressed as a string literal—matches aa, bb, and cc:

```
"(a|b|c)\\1"
```

This expression first matches a, b, or c. Whatever is matched is *tagged*. The regex pattern must then immediately match this tagged character again if it is to match the overall expression.

It will therefore match aa and bb, but not ab.

The next example is more practical: It finds a repeated word, in which a single space separates the two words:

```
"([A-Za-z]+) \\1"
```

This expression says, "Match a series of one or more letters. Tag the characters in this group. Then match a space. Finally, match an exact recurrence of the tagged characters." The following strings would therefore be matched:

```
"the the"
"Monday Monday"
"Rabbit Rabbit"
```

20.3.4. Character Classes

The C++11 grammar also provides a series of character classes that can be used to help specify a

range. For example, the following expression specifies a range consisting of any letter:

```
[[:alpha:]]
```

This is equivalent to:

```
[A-Za-z]
```

The following expression specifies a range consisting of any letter or punctuation character:

```
[[:alpha:][:punct:]]
```

Descriptions of the character classes follow.

[:alnum:]
Any letter or digit.

[:alpha:]
Any letter.

[:blank:]
A space or tab character.

[:cntrl:]
Any control character. (These are not printable.)

[:digit:]
Any decimal digit.

[:graph:]
Any printable character that is not a whitespace.

[:lower:]
Any lowercase letter.

[:print:]
Any printable character, including whitespaces.

[:punct:]
Any punctuation character.

[:space:]
A whitespace character, such as a blank space, tab, or newline.

[:upper:]
Any uppercase letter.

[:xdigit:]
A hexadecimal digit: This includes digits, as well as uppercase and lowercase letters.