

Regular expressions (C++)

07/15/2021 • 23 minutes to read •  +7

In this article

[Regular expression grammar](#)

[Grammar summary](#)

[Semantic details](#)

[Matching and searching](#)

[Format flags](#)

[See also](#)

The C++ standard library supports multiple regular expression grammars. This topic discusses the grammar variations available when using regular expressions.

Regular expression grammar

The regular expression grammar to use is by specified by the use of one of the `std::regex_constants::syntax_option_type` enumeration values. These regular expression grammars are defined in `std::regex_constants`:

- **ECMAScript**: This is closest to the grammar used by JavaScript and the .NET languages.
- **basic**: The POSIX basic regular expressions or BRE.
- **extended**: The POSIX extended regular expressions or ERE.
- **awk**: This is extended, but it has more escapes for non-printing characters.
- **grep**: This is basic, but it also allows newline (`\n`) characters to separate alternations.
- **egrep**: This is extended, but it also allows newline characters to separate alternations.

By default, if no grammar is specified, ECMAScript is assumed. Only one grammar may be specified.

Several flags can also be applied:

- **icase**: Ignore case when matching.
- **nosubs**: Ignore marked matches (that is, expressions in parentheses); no substitutions are stored.

- `optimize`: Make matching faster, at the possible expense of greater construction time.
- `collate`: Use locale-sensitive collation sequences (for example, ranges of the form `[a-z]`).

Zero or more flags may be combined with the grammar to specify the regular expression engine behavior. If only flags are specified, ECMAScript is assumed as the grammar.

Element

An element can be one of the following:

- An *ordinary character* that matches the same character in the target sequence.
- A *wildcard character* `.` that matches any character in the target sequence except a newline.
- A *bracket expression* of the form `[expr]`, which matches a character or a collation element in the target sequence that is also in the set defined by the expression `expr`, or of the form `[^expr]`, which matches a character or a collation element in the target sequence that isn't in the set defined by the expression `expr`.

The expression `expr` can contain any combination of the following:

- An individual character. Adds the character to the set defined by `expr`.
- A *character range* of the form `ch1-ch2`. Adds the characters that are represented by values in the closed range `[ch1, ch2]` to the set defined by `expr`.
- A *character class* of the form `[:name:]`. Adds the characters in the named class to the set defined by `expr`.
- An *equivalence class* of the form `[=e1t=]`. Adds the collating elements that are equivalent to `e1t` to the set defined by `expr`.
- A *collating symbol* of the form `[.e1t.]`. Adds the collation element `e1t` to the set defined by `expr`.
- An *anchor*. Anchor `^` matches the beginning of the target sequence. Anchor `$` matches the end of the target sequence.

A *capture group* of the form *(subexpression)*, or *\(subexpression\)* in basic and grep, which matches the sequence of characters in the target sequence that is matched by the pattern between the delimiters.

- An *identity escape* of the form *\k*, which matches the character *k* in the target sequence.

Examples:

- *a* matches the target sequence "a" but doesn't match the target sequences "B", "b", or "c".
- *.* matches all the target sequences "a", "B", "b", and "c".
- *[b-z]* matches the target sequences "b" and "c" but doesn't match the target sequences "a" or "B".
- *[:lower:]* matches the target sequences "a", "b", and "c" but doesn't match the target sequence "B".
- *(a)* matches the target sequence "a" and associates capture group 1 with the subsequence "a", but doesn't match the target sequences "B", "b", or "c".

In ECMAScript, basic, and grep, an element can also be a *back reference* of the form *\dd*, where *dd* represents a decimal value *N* that matches a sequence of characters in the target sequence that is the same as the sequence of characters that is matched by the *N*th *capture group*.

For example, *(a)\1* matches the target sequence "aa" because the first (and only) capture group matches the initial sequence "a" and then the *\1* matches the final sequence "a".

In ECMAScript, an element can also be one of the following:

- A *non-capture group* of the form *(?: subexpression)*. Matches the sequence of characters in the target sequence that is matched by the pattern between the delimiters.
- A limited *file format escape* of the form *\f*, *\n*, *\r*, *\t*, or *\v*. These match a form feed, newline, carriage return, horizontal tab, and vertical tab, respectively, in the target sequence.

- A *positive assert* of the form `(= subexpression)`. Matches the sequence of characters in the target sequence that is matched by the pattern between the delimiters, but doesn't change the match position in the target sequence.
- A *negative assert* of the form `(! subexpression)`. Matches any sequence of characters in the target sequence that doesn't match the pattern between the delimiters, and doesn't change the match position in the target sequence.
- A *hexadecimal escape sequence* of the form `\xhh`. Matches a character in the target sequence that is represented by the two hexadecimal digits `hh`.
- A *unicode escape sequence* of the form `\uhhhh`. Matches a character in the target sequence that is represented by the four hexadecimal digits `hhhh`.
- A *control escape sequence* of the form `\ck`. Matches the control character that is named by the character `k`.
- A *word boundary assert* of the form `\b`. Matches when the current position in the target sequence is immediately after a *word boundary*.
- A *negative word boundary assert* of the form `\B`. Matches when the current position in the target sequence isn't immediately after a *word boundary*.
- A *dsw character escape* of the form `\d`, `\D`, `\s`, `\S`, `\w`, `\W`. Provides a short name for a character class.

Examples:

- `(?:a)` matches the target sequence `"a"`, but `"(?:a)\1"` is invalid because there's no capture group 1.
- `(=a)a` matches the target sequence `"a"`. The positive assert matches the initial sequence `"a"` in the target sequence and the final `"a"` in the regular expression matches the initial sequence `"a"` in the target sequence.
- `(!a)a` doesn't match the target sequence `"a"`.
- `a\b`. matches the target sequence `"a~"`, but doesn't match the target sequence `"ab"`.
- `a\B`. matches the target sequence `"ab"`, but doesn't match the target sequence `"a~"`.

In awk, an element can also be one of the following:

- A *file format escape* of the form `\\`, `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, or `\v`. These match a backslash, alert, backspace, form feed, newline, carriage return, horizontal tab, and vertical tab, respectively, in the target sequence.
- An *octal escape sequence* of the form `\ooo`. Matches a character in the target sequence whose representation is the value represented by the one, two, or three octal digits `ooo`.

Repetition

Any element other than a *positive assert*, a *negative assert*, or an *anchor* can be followed by a repetition count. The most general kind of repetition count takes the form `{min,max}`, or `{min,max\}` in basic and grep. An element that is followed by this form of repetition count matches at least *min* successive occurrences and no more than *max* successive occurrences of a sequence that matches the element.

For example, `a{2,3}` matches the target sequence "aa" and the target sequence "aaa", but not the target sequence "a" or the target sequence "aaaa".

A repetition count can also take one of the following forms:

- `{min}` or `\{min}` in basic and grep. Equivalent to `{min,min}`.
- `{min,}` or `\{min,\}` in basic and grep. Equivalent to `{min,unbounded}`.
- `*` is equivalent to `{0,unbounded}`.

Examples:

- `a{2}` matches the target sequence "aa" but not the target sequence "a" or the target sequence "aaa".
- `a{2,}` matches the target sequence "aa", the target sequence "aaa", and so on, but doesn't match the target sequence "a".
- `a*` matches the target sequence "", the target sequence "a", the target sequence "aa", and so on.

For all grammars except basic and grep, a repetition count can also take one of the following forms:

- `?` is equivalent to `{0,1}`.
- `+` is equivalent to `{1,unbounded}`.

Examples:

- `a?` matches the target sequence `""` and the target sequence `"a"`, but not the target sequence `"aa"`.
- `a+` matches the target sequence `"a"`, the target sequence `"aa"`, and so on, but not the target sequence `""`.

In ECMAScript, all the forms of repetition count can be followed by the character `?` which designates a *non-greedy repetition*.

Concatenation

Regular expression elements, with or without *repetition counts*, can be concatenated to form longer regular expressions. The resulting expression matches a target sequence that is a concatenation of the sequences that are matched by the individual elements.

For example, `a{2,3}b` matches the target sequence `"aab"` and the target sequence `"aaab"`, but doesn't match the target sequence `"ab"` or the target sequence `"aaaab"`.

Alternation

In all regular expression grammars except basic and `grep`, a concatenated regular expression can be followed by the character `|` (pipe) and another concatenated regular expression. Any number of concatenated regular expressions can be combined in this manner. The resulting expression matches any target sequence that matches one or more of the concatenated regular expressions.

When more than one of the concatenated regular expressions match the target sequence, ECMAScript chooses the first of the concatenated regular expressions that matches the sequence as the match, which will be referred to as the *first match*. The other regular expression grammars choose the one that achieves the *longest match*.

For example, `ab|cd` matches the target sequence `"ab"` and the target sequence `"cd"`, but doesn't match the target sequence `"abd"` or the target sequence `"acd"`.

In `grep` and `egrep`, a newline character (`\n`) can be used to separate alternations.

Subexpression

In `basic` and `grep`, a subexpression is a concatenation. In the other regular expression grammars, a subexpression is an alternation.

Grammar summary

The following table summarizes the features that are available in the various regular expression grammars:

Element	basic	extended	ECMAScript	grep	egrep	awk
alternation using <code> </code>		+	+		+	+
alternation using <code>\n</code>				+	+	
anchor	+	+	+	+	+	+
back reference	+		+	+		
bracket expression	+	+	+	+	+	+
capture group using <code>()</code>		+	+		+	+
capture group using <code>\(\\)</code>	+			+		
control escape sequence			+			
dsw character escape			+			
file format escape			+			+
hexadecimal escape sequence			+			
identity escape	+	+	+	+	+	+

Element	basic	extended	ECMAScript	grep	egrep	awk
negative assert			+			
negative word boundary assert			+			
non-capture group			+			
non-greedy repetition			+			
octal escape sequence						+
ordinary character	+	+	+	+	+	+
positive assert			+			
repetition using {}		+	+		+	+
repetition using \{\}	+			+		
repetition using *	+	+	+	+	+	+
repetition using ? and +		+	+		+	+
unicode escape sequence			+			
wildcard character	+	+	+	+	+	+
word boundary assert			+			

Semantic details

Anchor

An anchor matches a position in the target string, not a character. A `^` matches the beginning of the target string, and a `$` matches the end of the target string.

Back reference

A back reference is a backslash that is followed by a decimal value *N*. It matches the contents of the *N*th *capture group*. The value of *N* must not be more than the number of capture groups that precede the back reference. In basic and `grep`, the value of *N* is determined by the decimal digit that follows the backslash. In ECMAScript, the value of *N* is determined by all the decimal digits that immediately follow the backslash. Therefore, in basic and `grep`, the value of *N* is never more than 9, even if the regular expression has more than nine capture groups. In ECMAScript, the value of *N* is unbounded.

Examples:

- `((a+)(b+))(c+)\3` matches the target sequence "aabbcbcb". The back reference `\3` matches the text in the third capture group, that is, the "(b+)". It doesn't match the target sequence "aabbcbcb".
- `(a)\2` isn't valid.
- `(b((((((((a))))))))))\10` has different meanings in basic and in ECMAScript. In basic, the back reference is `\1`. The back reference matches the contents of the first capture group (that is, the one that begins with `(b` and ends with the final `)` and comes before the back reference), and the final `0` matches the ordinary character `0`. In ECMAScript, the back reference is `\10`. It matches the tenth capture group, that is, the innermost one.

Bracket expression

A bracket expression defines a set of characters and *collating elements*. When the bracket expression begins with the character `^` the match succeeds if no elements in the set match the current character in the target sequence. Otherwise, the match succeeds if any one of the elements in the set matches the current character in the target sequence.

The set of characters can be defined by listing any combination of *individual characters*, *character ranges*, *character classes*, *equivalence classes*, and *collating symbols*.

Capture group

A capture group marks its contents as a single unit in the regular expression grammar and labels the target text that matches its contents. The label that is associated with each

capture group is a number, which is determined by counting the opening parentheses that mark capture groups up to and including the opening parenthesis that marks the current capture group. In this implementation, the maximum number of capture groups is 31.

Examples:

- `ab+` matches the target sequence `"abb"`, but doesn't match the target sequence `"abab"`.
- `(ab)+` doesn't match the target sequence `"abb"`, but matches the target sequence `"abab"`.
- `((a+)(b+))(c+)` matches the target sequence `"aabbbc"` and associates capture group 1 with the subsequence `"aabb"`, capture group 2 with the subsequence `"aa"`, capture group 3 with `"bbb"`, and capture group 4 with the subsequence `"c"`.

Character class

A character class in a bracket expression adds all the characters in the named class to the character set that is defined by the bracket expression. To create a character class, use `[:` followed by the name of the class, followed by `:]`.

Internally, names of character classes are recognized by calling `id = traits.lookup_classname`. A character `ch` belongs to such a class if `traits.isctype(ch, id)` returns true. The default `regex_traits` template supports the class names in the following table.

Class Name	Description
<code>alnum</code>	lowercase letters, uppercase letters, and digits
<code>alpha</code>	lowercase letters and uppercase letters
<code>blank</code>	space or tab
<code>cntrl</code>	the <i>file format escape</i> characters
<code>digit</code>	digits
<code>graph</code>	lowercase letters, uppercase letters, digits, and punctuation

Class Name	Description
<code>lower</code>	lowercase letters
<code>print</code>	lowercase letters, uppercase letters, digits, punctuation, and space
<code>punct</code>	punctuation
<code>space</code>	space
<code>upper</code>	uppercase characters
<code>xdigit</code>	digits, <code>a</code> , <code>b</code> , <code>c</code> , <code>d</code> , <code>e</code> , <code>f</code> , <code>A</code> , <code>B</code> , <code>C</code> , <code>D</code> , <code>E</code> , <code>F</code>
<code>d</code>	same as <code>digit</code>
<code>s</code>	same as <code>space</code>
<code>w</code>	same as <code>alnum</code>

Character range

A character range in a bracket expression adds all the characters in the range to the character set that is defined by the bracket expression. To create a character range, put the character `'-'` between the first and last characters in the range. A character range puts all characters that have a numeric value that is more than or equal to the numeric value of the first character, and less than or equal to the numeric value of the last character, into the set. Notice that this set of added characters depends on the platform-specific representation of characters. If the character `'-'` occurs at the beginning or the end of a bracket expression, or as the first or last character of a character range, it represents itself.

Examples:

- `[0-7]` represents the set of characters `{ 0, 1, 2, 3, 4, 5, 6, 7 }`. It matches the target sequences `"0"`, `"1"`, and so on, but not `"a"`.
- On systems that use ASCII character encoding, `[h-k]` represents the set of characters `{ h, i, j, k }`. It matches the target sequences `"h"`, `"i"`, and so on, but not `"\x8A"` or `"0"`.

- On systems that use EBCDIC character encoding, `[h-k]` represents the set of characters `{ h, i, '\x8A', '\x8B', '\x8C', '\x8D', '\x8E', '\x8F', '\x90', j, k }` (`h` is encoded as `0x88` and `k` is encoded as `0x92`). It matches the target sequences `"h"`, `"i"`, `"\x8A"`, and so on, but not `"0"`.
- `[-0-24]` represents the set of characters `{ -, 0, 1, 2, 4 }`.
- `[0-2-]` represents the set of characters `{ 0, 1, 2, - }`.
- On systems that use ASCII character encoding, `[+--]` represents the set of characters `{ +, -, }`.

However, when locale-sensitive ranges are used, the characters in a range are determined by the collation rules for the locale. Characters that collate after the first character in the definition of the range and before the last character in the definition of the range are in the set. The two end characters are also in the set.

Collating element

A collating element is a multi-character sequence that is treated as a single character.

Collating symbol

A collating symbol in a bracket expression adds a *collating element* to the set that is defined by the bracket expression. To create a collating symbol, use `[.` followed by the collating element, followed by `.]`

Control escape sequence

A control escape sequence is a backslash followed by the letter `'c'` followed by one of the letters `'a'` through `'z'` or `'A'` through `'Z'`. It matches the ASCII control character that is named by that letter. For example, `"\ci"` matches the target sequence `"\x09"`, because `Ctrl+I` has the value `0x09`.

DSW character escape

A dsw character escape is a short name for a character class, as shown in the following table.

Escape Sequence	Equivalent Named Class	Default Named Class
<code>\d</code>	<code>[[:d:]]</code>	<code>[[:digit:]]</code>
<code>\D</code>	<code>[^ :d:]</code>	<code>[^ :digit:]</code>
<code>\s</code>	<code>[[:s:]]</code>	<code>[[:space:]]</code>
<code>\S</code>	<code>[^ :s:]</code>	<code>[^ :space:]</code>
<code>\w</code>	<code>[[:w:]]</code>	<code>[a-zA-Z0-9_]*</code>
<code>\W</code>	<code>[^ :w:]</code>	<code>[^a-zA-Z0-9_]*</code>

*ASCII character set

Equivalence class

An equivalence class in a bracket expression adds all the characters and *collating elements* that are equivalent to the collating element in the equivalence class definition to the set that is defined by the bracket expression.

To create an equivalence class, use [= followed by a collating element followed by =].

Internally, two collating elements `elt1` and `elt2` are equivalent if

```
traits.transform_primary(elt1.begin(), elt1.end()) ==
traits.transform_primary(elt2.begin(), elt2.end()).
```

File format escape

A file format escape consists of the usual C language character escape sequences, `\\`, `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`. These have the usual meanings, that is, backslash, alert, backspace, form feed, newline, carriage return, horizontal tab, and vertical tab, respectively. In ECMAScript, `\a` and `\b` aren't allowed. (`\\` is allowed, but it's an identity escape, not a file format escape).

Hexadecimal escape sequence

A hexadecimal escape sequence is a backslash followed by the letter `x` followed by two hexadecimal digits (`0-9a-fA-F`). It matches a character in the target sequence that has the value that is specified by the two digits.

For example, `"\x41"` matches the target sequence `"a"` when ASCII character encoding is used.

Identity escape

An identity escape is a backslash followed by a single character. It matches that character. It's required when the character has a special meaning. Using the identity escape removes the special meaning. For example:

- `a*` matches the target sequence `"aaa"`, but doesn't match the target sequence `"a*"`.
- `a*` doesn't match the target sequence `"aaa"`, but matches the target sequence `"a*"`.

The set of characters that are allowed in an identity escape depends on the regular expression grammar, as shown in the following table.

Grammar	Allowed Identity Escape Characters
basic, grep	{ () { } . [\ * ^ \$ }
extended, egrep	{ () { . [\ * ^ \$ + ? }
awk, extended	plus { " / }
ECMAScript	All characters except those that can be part of an identifier. Typically, this includes letters, digits, <code>\$</code> , <code>_</code> , and unicode escape sequences. For more information, see the ECMAScript Language Specification.

Individual character

An individual character in a bracket expression adds that character to the character set that is defined by the bracket expression. Anywhere in a bracket expression except at the beginning, a `^` represents itself.

Examples:

- `[abc]` matches the target sequences "a", "b", and "c", but not the sequence "d".
- `^[abc]` matches the target sequence "d", but not the target sequences "a", "b", or "c".
- `[a^bc]` matches the target sequences "a", "b", "c", and "^", but not the target sequence "d".

In all regular expression grammars except ECMAScript, if a `]` is the first character that follows the opening `[` or is the first character that follows an initial `^`, it represents itself.

Examples:

- `[]a` is invalid because there's no `]` to end the bracket expression.
- `[]abc` matches the target sequences "a", "b", "c", and "]", but not the target sequence "d".
- `[^]abc` matches the target sequence "d", but not the target sequences "a", "b", "c", or "]".

In ECMAScript, use `\]` to represent the character `]` in a bracket expression.

Examples:

- `[]a` matches the target sequence "a" because the bracket expression is empty.
- `[\]abc` matches the target sequences "a", "b", "c", and "]" but not the target sequence "d".

Negative assert

A negative assert matches anything but its contents. It doesn't consume any characters in the target sequence.

For example, `(!aa)(a*)` matches the target sequence "a" and associates capture group 1 with the subsequence "a". It doesn't match the target sequence "aa" or the target sequence "aaa".

Negative word boundary assert

A negative word boundary assert matches if the current position in the target string isn't immediately after a *word boundary*.

Non-capture group

A non-capture group marks its contents as a single unit in the regular expression grammar, but doesn't label the target text.

For example, `(a)(?:b)*(c)` matches the target text "abbc" and associates capture group 1 with the subsequence "a" and capture group 2 with the subsequence "c".

Non-greedy repetition

A non-greedy repetition consumes the shortest subsequence of the target sequence that matches the pattern. A greedy repetition consumes the longest. For example, `(a+)(a*b)` matches the target sequence "aaab".

When a non-greedy repetition is used, it associates capture group 1 with the subsequence "a" at the beginning of the target sequence and capture group 2 with the subsequence "aab" at the end of the target sequence.

When a greedy match is used, it associates capture group 1 with the subsequence "aaa" and capture group 2 with the subsequence "b".

Octal escape sequence

An octal escape sequence is a backslash followed by one, two, or three octal digits (0-7). It matches a character in the target sequence that has the value that is specified by those digits. If all the digits are 0, the sequence is invalid.

For example, `\101` matches the target sequence "a" when ASCII character encoding is used.

Ordinary character

An ordinary character is any valid character that doesn't have a special meaning in the current grammar.

In ECMAScript, the following characters have special meanings:

- `^ $ \ . * + ? () [] { } |`

In basic and grep, the following characters have special meanings:

- `. [\`

Also in basic and grep, the following characters have special meanings when they're used in a particular context:

- `*` has a special meaning in all cases except when it's the first character in a regular expression or the first character that follows an initial `^` in a regular expression, or when it's the first character of a capture group or the first character that follows an initial `^` in a capture group.
- `^` has a special meaning when it's the first character of a regular expression.
- `$` has a special meaning when it's the last character of a regular expression.

In extended, egrep, and awk, the following characters have special meanings:

- `. [\ (* + ? { |`

Also in extended, egrep, and awk, the following characters have special meanings when they're used in a particular context.

- `)` has a special meaning when it matches a preceding `(`
- `^` has a special meaning when it's the first character of a regular expression.
- `$` has a special meaning when it's the last character of a regular expression.

An ordinary character matches the same character in the target sequence. By default, this means that the match succeeds if the two characters are represented by the same value. In a case-insensitive match, two characters `ch0` and `ch1` match if

`traits.translate_nocase(ch0) == traits.translate_nocase(ch1)`. In a locale-sensitive match, two characters `ch0` and `ch1` match if `traits.translate(ch0) == traits.translate(ch1)`.

Positive assert

A positive assert matches its contents, but doesn't consume any characters in the target sequence.

Examples:

- `(=aa)(a*)` matches the target sequence "aaaa" and associates capture group 1 with the subsequence "aaaa".
- `(aa)(a*)` matches the target sequence "aaaa" and associates capture group 1 with the subsequence "aa" at the beginning of the target sequence and capture group 2 with the subsequence "aa" at the end of the target sequence.
- `(=aa)(a)|(a)` matches the target sequence "a" and associates capture group 1 with an empty sequence (because the positive assert failed) and capture group 2 with the subsequence "a". It also matches the target sequence "aa" and associates capture group 1 with the subsequence "aa" and capture group 2 with an empty sequence.

Unicode escape sequence

A unicode escape sequence is a backslash followed by the letter 'u' followed by four hexadecimal digits (0-9a-fA-F). It matches a character in the target sequence that has the value that is specified by the four digits. For example, `\u0041` matches the target sequence "a" when ASCII character encoding is used.

Wildcard character

A wildcard character matches any character in the target expression except a newline.

Word boundary

A word boundary occurs in the following situations:

- The current character is at the beginning of the target sequence and is one of the word characters `A-Za-z0-9_`
- The current character position is past the end of the target sequence and the last character in the target sequence is one of the word characters.

- The current character is one of the word characters and the preceding character isn't.
- The current character isn't one of the word characters and the preceding character is.

Word boundary assert

A word boundary assert matches when the current position in the target string is immediately after a *word boundary*.

Matching and searching

For a regular expression to match a target sequence, the entire regular expression must match the entire target sequence. For example, the regular expression `bcd` matches the target sequence `"bcd"` but doesn't match the target sequence `"abcd"` nor the target sequence `"bcde"`.

For a regular expression search to succeed, there must be a subsequence somewhere in the target sequence that matches the regular expression. The search typically finds the left-most matching subsequence.

Examples:

- A search for the regular expression `bcd` in the target sequence `"bcd"` succeeds and matches the entire sequence. The same search in the target sequence `"abcd"` also succeeds and matches the last three characters. The same search in the target sequence `"bcde"` also succeeds and matches the first three characters.
- A search for the regular expression `bcd` in the target sequence `"bcdbcd"` succeeds and matches the first three characters.

If there's more than one subsequence that matches at some location in the target sequence, there are two ways to choose the matching pattern.

First match chooses the subsequence that was found first when the regular expression is matched.

Longest match chooses the longest subsequence from the ones that match at that location. If there's more than one subsequence that has the maximal length, longest match chooses the one that was found first.

For example, when first match is used, a search for the regular expression `b|bc` in the target sequence `"abcd"` matches the subsequence `"b"` because the left-hand term of the alternation matches that subsequence; therefore, first match doesn't try the right-hand term of the alternation. When longest match is used, the same search matches `"bc"` because `"bc"` is longer than `"b"`.

A partial match succeeds if the match reaches the end of the target sequence without failing, even if it hasn't reached the end of the regular expression. Therefore, after a partial match succeeds, appending characters to the target sequence could cause a later partial match to fail. However, after a partial match fails, appending characters to the target sequence can't cause a later partial match to succeed. For example, with a partial match, `ab` matches the target sequence `"a"` but not `"ac"`.

Format flags



ECMAScript Format Rules	sed Format Rules	Replacement Text
<code>&</code>	<code>&</code>	The character sequence that matches the entire regular expression: <code>[match[0].first, match[0].second)</code>
<code>\$\$</code>		<code>\$</code>
	<code>\&</code>	<code>&</code>
<code>\$`</code> (dollar sign followed by back quote)		The character sequence that precedes the subsequence that matches the regular expression: <code>[match.prefix().first, match.prefix().second)</code>
<code>\$'</code> (dollar sign followed by forward quote)		The character sequence that follows the subsequence that matches the regular expression: <code>[match.suffix().first, match.suffix().second)</code>
<code>\$n</code>	<code>\n</code>	The character sequence that matches the capture group at position <code>n</code> , where <code>n</code> is a number between 0 and 9: <code>[match[n].first, match[n].second)</code>
	<code>\\n</code>	<code>\n</code>

ECMAScript Format Rules	sed Format Rules	Replacement Text
	\$nn	The character sequence that matches the capture group at position nn, where nn is a number between 10 and 99: <code>[match[nn].first, match[nn].second)</code>

See also

[C++ Standard Library Overview](#)

Is this page helpful?

 Yes  No

Recommended content

[<iostream>](#)

Learn more about: [<iostream>](#)

[C++ standard library header files](#)

C++ standard library header files, categorized

[<vector>](#)

Learn more about: [<vector>](#)

[Functions with Variable Argument Lists \(C++\)](#)

Learn more about: Functions with Variable Argument Lists (C++)

Show more 