NAME

pcregrep - a grep with Perl-compatible regular expressions.

SYNOPSIS

pcregrep [options] [long options] [pattern] [path1 path2 ...]

DESCRIPTION

pcregrep searches files for character patterns, in the same way as other grep commands do, but it uses the PCRE regular expression library to support patterns that are compatible with the regular expressions of Perl 5. See **pcrepattern**(3) for a full description of syntax and semantics of the regular expressions that PCRE supports.

Patterns, whether supplied on the command line or in a separate file, are given without delimiters. For example:

```
  pcregrep Thursday /etc/motd
```

If you attempt to use delimiters (for example, by surrounding a pattern with slashes, as is common in Perl scripts), they are interpreted as part of the pattern. Quotes can of course be used on the command line because they are interpreted by the shell, and indeed they are required if a pattern contains white space or shell metacharacters.

The first argument that follows any option settings is treated as the single pattern to be matched when neither **-e** nor **-f** is present. Conversely, when one or both of these options are used to specify patterns, all arguments are treated as path names. At least one of **-e**, **-f**, or an argument pattern must be provided.

If no files are specified, **pcregrep** reads the standard input. The standard input can also be referenced by a name consisting of a single hyphen. For example:

```
  pcregrep some-pattern /file1 - /file3
```

By default, each line that matches the pattern is copied to the standard output, and if there is more than one file, the file name is output at the start of each line. However, there are options that can change how **pcregrep** behaves. In particular, the **-M** option makes it possible to search for patterns that span line boundaries. What defines a line boundary is controlled by the **-N** (**--newline**) option.

Patterns are limited to 8K or BUFSIZ characters, whichever is the greater. BUFSIZ is defined in **<stdio.h>**.

If the **LC_ALL** or **LC_CTYPE** environment variable is set, **pcregrep** uses the value to set a locale when calling the PCRE library. The **--locale** option can be used to override this.

OPTIONS

--
This terminate the list of options. It is useful if the next item on the command line starts with a hyphen but is not an option. This allows for the processing of patterns and filenames that start with hyphens.

-A *number*, **--after-context=***number*
Output *number* lines of context after each matching line. If filenames and/or line numbers are being output, a hyphen separator is used instead of a colon for the context lines. A line containing "--" is output between each group of lines, unless they are in fact contiguous in the input file. The value of *number* is expected to be relatively small. However, **pcregrep** guarantees to have up to 8K of following text available for context output.

-B *number*, **--before-context=***number*
Output *number* lines of context before each matching line. If filenames and/or line numbers are being output, a hyphen separator is used instead of a colon for the context lines. A line containing "--" is output between each group of lines, unless they are in fact contiguous in the input file. The value of *number* is expected to be relatively small. However, **pcregrep** guarantees to have up to 8K of preceding text available for context output.

-C *number*, **--context=***number*
Output *number* lines of context both before and after each matching line. This is equivalent to setting both **-A** and **-B** to the same value.

-c, **--count** Do not output individual lines; instead just output a count of the number of lines that would otherwise have been output. If several files are given, a count is output for each of them. In this mode, the **-A**, **-B**, and **-C** options are ignored.

--colour, **--color**
If this option is given without any data, it is equivalent to "--colour=auto". If data is required, it must be given in the same shell item, separated by an equals sign.

--colour=*value*, **--color=***value*
This option specifies under what circumstances the part of a line that matched a pattern should be coloured in the output. The value may be "never" (the default), "always", or "auto". In the latter case, colouring happens only if the standard output is connected to a terminal. The colour can be specified by setting the environment variable PCREGREP_COLOUR or PCREGREP_COLOR. The value of this variable should be a string of two numbers, separated by a semicolon. They are copied directly into the control string for setting colour on a terminal, so it is your responsibility to ensure that they make sense. If neither of the environment variables is set, the default is "1;31", which gives red.

-D *action*, **--devices=***action*
If an input path is not a regular file or a directory, "action" specifies how it is to be processed. Valid values are "read" (the default) or "skip" (silently skip the path).

-d *action*, **--directories=***action*
If an input path is a directory, "action" specifies how it is to be processed. Valid values are "read" (the default), "recurse" (equivalent to the **-r** option), or "skip" (silently skip the path). In the default case, directories are read as if they were ordinary files. In some operating systems the effect of reading a directory like this is an immediate end-of-file.

-e *pattern*, **--regex=***pattern*,
--regexp=*pattern* Specify a pattern to be matched. This option can be used multiple times in order to specify several patterns. It can also be used as a way of specifying a single pattern that starts with a hyphen. When **-e** is used, no argument pattern is taken from the command line; all arguments are treated as file names. There is an overall maximum of 100 patterns. They are applied to each line in the order in which they are defined until one matches (or fails to match if **-v** is used). If **-f** is used with **-e**, the command line patterns are matched first, followed by the patterns from the file, independent of the order in which these options are specified. Note that multiple use of **-e** is not the same as a single pattern with alternatives. For example, X|Y finds the first character in a line that is X or Y, whereas if the two patterns are given separately, **pcregrep** finds X if it is present, even if it follows Y in the line. It finds Y only if there is no X in the line. This really matters only if you are using **-o** to show the portion of the line that matched.

--exclude=*pattern*
When **pcregrep** is searching the files in a directory as a consequence of the **-r** (recursive search) option, any files whose names match the pattern are excluded. The pattern is a PCRE regular expression. If a file name matches both **--include** and **--exclude**, it is excluded. There is no short form for this option.

-F, **--fixed-strings**
Interpret each pattern as a list of fixed strings, separated by newlines, instead of as a regular expression. The **-w** (match as a word) and **-x** (match whole line) options can be used with **-F**. They apply to each of the fixed strings. A line is selected if any of the fixed strings are found in it (subject to **-w** or **-x**, if present).

-f *filename*, **--file=***filename*
Read a number of patterns from the file, one per line, and match them against each line of input. A data line is output if any of the patterns match it. The filename can be given as "-" to refer to the standard input. When **-f** is used, patterns specified on the command line using **-e** may also be present; they are tested before the file's patterns. However, no other pattern is taken from the command line; all arguments are treated as file names. There is an

overall maximum of 100 patterns. Trailing white space is removed from each line, and blank lines are ignored. An empty file contains no patterns and therefore matches nothing.

-H, **--with-filename**
Force the inclusion of the filename at the start of output lines when searching a single file. By default, the filename is not shown in this case. For matching lines, the filename is followed by a colon and a space; for context lines, a hyphen separator is used. If a line number is also being output, it follows the file name without a space.

-h, **--no-filename**
Suppress the output filenames when searching multiple files. By default, filenames are shown when multiple files are searched. For matching lines, the filename is followed by a colon and a space; for context lines, a hyphen separator is used. If a line number is also being output, it follows the file name without a space.

--help Output a brief help message and exit.

-i, **--ignore-case**
Ignore upper/lower case distinctions during comparisons.

--include=*pattern*
When **pcregrep** is searching the files in a directory as a consequence of the **-r** (recursive search) option, only those files whose names match the pattern are included. The pattern is a PCRE regular expression. If a file name matches both **--include** and **--exclude**, it is excluded. There is no short form for this option.

-L, **--files-without-match**
Instead of outputting lines from the files, just output the names of the files that do not contain any lines that would have been output. Each file name is output once, on a separate line.

-l, **--files-with-matches**
Instead of outputting lines from the files, just output the names of the files containing lines that would have been output. Each file name is output once, on a separate line. Searching stops as soon as a matching line is found in a file.

--label=*name*
This option supplies a name to be used for the standard input when file names are being output. If not supplied, "(standard input)" is used. There is no short form for this option.

--locale=*locale-name*
This option specifies a locale to be used for pattern matching. It overrides the value in the **LC_ALL** or **LC_CTYPE** environment variables. If no locale is specified, the PCRE library's default (usually the "C" locale) is used. There is no short form for this option.

-M, **--multiline**
Allow patterns to match more than one line. When this option is given, patterns may usefully contain literal newline characters and internal occurrences of ˆ and $ characters. The output for any one match may consist of more than one line. When this option is set, the PCRE library is called in "multiline" mode. There is a limit to the number of lines that can be matched, imposed by the way that **pcregrep** buffers the input file as it scans it. However, **pcregrep** ensures that at least 8K characters or the rest of the document (whichever is the shorter) are available for forward matching, and similarly the previous 8K characters (or all the previous characters, if fewer than 8K) are guaranteed to be available for lookbehind assertions.

-N *newline-type*, **--newline=***newline-type*
The PCRE library supports four different conventions for indicating the ends of lines. They are the single-character sequences CR (carriage return) and LF (linefeed), the two-character sequence CRLF, and an "any" convention, in which any Unicode line ending sequence is assumed to end a line. The Unicode sequences are the three just mentioned, plus VT (vertical tab, U+000B), FF (formfeed, U+000C), NEL (next line, U+0085), LS (line separator, U+2028), and PS (paragraph separator, U+0029).

When the PCRE library is built, a default line-ending sequence is specified. This is normally the standard sequence for the operating system. Unless otherwise specified by this option, **pcregrep** uses the library's default. The possible values for this option are CR, LF, CRLF, or ANY. This makes it possible to use **pcregrep** on files that have come from other environments without having to modify their line endings. If the data that is being scanned does not agree with the convention set by this option, **pcregrep** may behave in strange ways.

-n, **--line-number**
Precede each output line by its line number in the file, followed by a colon and a space for matching lines or a hyphen and a space for context lines. If the filename is also being output, it precedes the line number.

-o, **--only-matching**
Show only the part of the line that matched a pattern. In this mode, no context is shown. That is, the **-A**, **-B**, and **-C** options are ignored.

-q, **--quiet** Work quietly, that is, display nothing except error messages. The exit status indicates whether or not any matches were found.

-r, **--recursive**
If any given path is a directory, recursively scan the files it contains, taking note of any **--include** and **--exclude** settings. By default, a directory is read as a normal file; in some operating systems this gives an immediate end-of-file. This option is a shorthand for setting the **-d** option to "recurse".

-s, **--no-messages**
Suppress error messages about non-existent or unreadable files. Such files are quietly skipped. However, the return code is still 2, even if matches were found in other files.

-u, **--utf-8** Operate in UTF-8 mode. This option is available only if PCRE has been compiled with UTF-8 support. Both patterns and subject lines must be valid strings of UTF-8 characters.

-V, **--version**
Write the version numbers of **pcregrep** and the PCRE library that is being used to the standard error stream.

-v, **--invert-match**
Invert the sense of the match, so that lines which do *not* match any of the patterns are the ones that are found.

-w, **--word-regex**, **--word-regexp**
Force the patterns to match only whole words. This is equivalent to having \b at the start and end of the pattern.

-x, **--line-regex**, **--line-regexp**
Force the patterns to be anchored (each must start matching at the beginning of a line) and in addition, require them to match entire lines. This is equivalent to having ^ and $ characters at the start and end of each alternative branch in every pattern.

ENVIRONMENT VARIABLES

The environment variables **LC_ALL** and **LC_CTYPE** are examined, in that order, for a locale. The first one that is set is used. This can be overridden by the **--locale** option. If no locale is set, the PCRE library's default (usually the "C" locale) is used.

NEWLINES

The **-N** (**--newline**) option allows **pcregrep** to scan files with different newline conventions from the default. However, the setting of this option does not affect the way in which **pcregrep** writes information to the standard error and output streams. It uses the string "\n" in C **printf()** calls to indicate newlines, relying on the C I/O library to convert this to an appropriate sequence if the output is sent to a file.

OPTIONS COMPATIBILITY

The majority of short and long forms of **pcregrep**'s options are the same as in the GNU **grep** program.

Any long option of the form **--xxx-regexp** (GNU terminology) is also available as **--xxx-regex** (PCRE terminology). However, the **--locale**, **-M**, **--multiline**, **-u**, and **--utf-8** options are specific to **pcregrep**.

OPTIONS WITH DATA

There are four different ways in which an option with data can be specified. If a short form option is used, the data may follow immediately, or in the next command line item. For example:

```
  -f/some/file
  -f /some/file
```

If a long form option is used, the data may appear in the same command line item, separated by an equals character, or (with one exception) it may appear in the next command line item. For example:

```
  --file=/some/file
  --file /some/file
```

Note, however, that if you want to supply a file name beginning with ˜ as data in a shell command, and have the shell expand ˜ to a home directory, you must separate the file name from the option, because the shell does not treat ˜ specially unless it is at the start of an item.

The exception to the above is the **--colour** (or **--color**) option, for which the data is optional. If this option does have data, it must be given in the first form, using an equals character. Otherwise it will be assumed that it has no data.

MATCHING ERRORS

It is possible to supply a regular expression that takes a very long time to fail to match certain lines. Such patterns normally involve nested indefinite repeats, for example: (a+)*\d when matched against a line of a's with no final digit. The PCRE matching function has a resource limit that causes it to abort in these circumstances. If this happens, **pcregrep** outputs an error message and the line that caused the problem to the standard error stream. If there are more than 20 such errors, **pcregrep** gives up.

DIAGNOSTICS

Exit status is 0 if any matches were found, 1 if no matches were found, and 2 for syntax errors and non-existent or inacessible files (even if matches were found in other files) or too many matching errors. Using the **-s** option to suppress error messages about inaccessble files does not affect the return code.

SEE ALSO

pcrepattern(3), **pcretest**(1).

AUTHOR

Philip Hazel
University Computing Service
Cambridge CB2 3QH, England.

Last updated: 29 November 2006

NAME

pcretest - a program for testing Perl-compatible regular expressions.

SYNOPSIS

pcretest [options] [source] [destination]

pcretest was written as a test program for the PCRE regular expression library itself, but it can also be used for experimenting with regular expressions. This document describes the features of the test program; for details of the regular expressions themselves, see the **pcrepattern** documentation. For details of the PCRE library function calls and their options, see the **pcreapi** documentation.

OPTIONS

-b	Behave as if each regex has the **/B** (show bytecode) modifier; the internal form is output after compilation.
-C	Output the version number of the PCRE library, and all available information about the optional features that are included, and then exit.
-d	Behave as if each regex has the **/D** (debug) modifier; the internal form and information about the compiled pattern is output after compilation; **-d** is equivalent to **-b -i**.
-dfa	Behave as if each data line contains the \D escape sequence; this causes the alternative matching function, **pcre_dfa_exec()**, to be used instead of the standard **pcre_exec()** function (more detail is given below).
-help	Output a brief summary these options and then exit.
-i	Behave as if each regex has the **/I** modifier; information about the compiled pattern is given after compilation.
-m	Output the size of each compiled pattern after it has been compiled. This is equivalent to adding **/M** to each regular expression. For compatibility with earlier versions of pcretest, **-s** is a synonym for **-m**.
-o *osize*	Set the number of elements in the output vector that is used when calling **pcre_exec()** or **pcre_dfa_exec()** to be *osize*. The default value is 45, which is enough for 14 capturing subexpressions for **pcre_exec()** or 22 different matches for **pcre_dfa_exec()**. The vector size can be changed for individual matching calls by including \O in the data line (see below).
-p	Behave as if each regex has the **/P** modifier; the POSIX wrapper API is used to call PCRE. None of the other options has any effect when **-p** is set.
-q	Do not output the version number of **pcretest** at the start of execution.
-S *size*	On Unix-like systems, set the size of the runtime stack to *size* megabytes.
-t	Run each compile, study, and match many times with a timer, and output resulting time per compile or match (in milliseconds). Do not set **-m** with **-t**, because you will then get the size output a zillion times, and the timing will be distorted. You can control the number of iterations that are used for timing by following **-t** with a number (as a separate item on the command line). For example, "-t 1000" would iterate 1000 times. The default is to iterate 500000 times.
-tm	This is like **-t** except that it times only the matching phase, not the compile or study phases.

DESCRIPTION

If **pcretest** is given two filename arguments, it reads from the first and writes to the second. If it is given only one filename argument, it reads from that file and writes to stdout. Otherwise, it reads from stdin and writes to stdout, and prompts for each line of input, using "re>" to prompt for regular expressions, and "data>" to prompt for data lines.

The program handles any number of sets of input on a single input file. Each set starts with a regular expression, and continues with any number of data lines to be matched against the pattern.

Each data line is matched separately and independently. If you want to do multi-line matches, you have to use the \n escape sequence (or \r or \r\n, etc., depending on the newline setting) in a single line of input to encode the newline sequences. There is no limit on the length of data lines; the input buffer is automatically extended if it is too small.

An empty line signals the end of the data lines, at which point a new regular expression is read. The regular expressions are given enclosed in any non-alphanumeric delimiters other than backslash, for example:

```
  /(a|bc)x+yz/
```

White space before the initial delimiter is ignored. A regular expression may be continued over several input lines, in which case the newline characters are included within it. It is possible to include the delimiter within the pattern by escaping it, for example

```
  /abc\/def/
```

If you do so, the escape and the delimiter form part of the pattern, but since delimiters are always non-alphanumeric, this does not affect its interpretation. If the terminating delimiter is immediately followed by a backslash, for example,

```
  /abc/\
```

then a backslash is added to the end of the pattern. This is done to provide a way of testing the error condition that arises if a pattern finishes with a backslash, because

```
  /abc\/
```

is interpreted as the first line of a pattern that starts with "abc/", causing pcretest to read the next line as a continuation of the regular expression.

PATTERN MODIFIERS

A pattern may be followed by any number of modifiers, which are mostly single characters. Following Perl usage, these are referred to below as, for example, "the **/i** modifier", even though the delimiter of the pattern need not always be a slash, and no slash is used when writing modifiers. Whitespace may appear between the final pattern delimiter and the first modifier, and between the modifiers themselves.

The **/i**, **/m**, **/s**, and **/x** modifiers set the PCRE_CASELESS, PCRE_MULTILINE, PCRE_DOTALL, or PCRE_EXTENDED options, respectively, when **pcre_compile()** is called. These four modifier letters have the same effect as they do in Perl. For example:

```
  /caseless/i
```

The following table shows additional modifiers for setting PCRE options that do not correspond to anything in Perl:

/A	PCRE_ANCHORED
/C	PCRE_AUTO_CALLOUT
/E	PCRE_DOLLAR_ENDONLY
/f	PCRE_FIRSTLINE
/J	PCRE_DUPNAMES
/N	PCRE_NO_AUTO_CAPTURE
/U	PCRE_UNGREEDY
/X	PCRE_EXTRA
/<cr>	PCRE_NEWLINE_CR
/<lf>	PCRE_NEWLINE_LF
/<crlf>	PCRE_NEWLINE_CRLF
/<any>	PCRE_NEWLINE_ANY

Those specifying line ending sequencess are literal strings as shown. This example sets multiline matching with CRLF as the line ending sequence:

```
 /^abc/m<crlf>
```

Details of the meanings of these PCRE options are given in the **pcreapi** documentation.

Finding all matches in a string

Searching for all possible matches within each subject string can be requested by the **/g** or **/G** modifier. After finding a match, PCRE is called again to search the remainder of the subject string. The difference between **/g** and **/G** is that the former uses the *startoffset* argument to **pcre_exec()** to start searching at a new point within the entire string (which is in effect what Perl does), whereas the latter passes over a shortened substring. This makes a difference to the matching process if the pattern begins with a lookbehind assertion (including \b or \B).

If any call to **pcre_exec()** in a **/g** or **/G** sequence matches an empty string, the next call is done with the PCRE_NOTEMPTY and PCRE_ANCHORED flags set in order to search for another, non-empty, match at the same point. If this second match fails, the start offset is advanced by one, and the normal match is retried. This imitates the way Perl handles such cases when using the **/g** modifier or the **split()** function.

Other modifiers

There are yet more modifiers for controlling the way **pcretest** operates.

The **/+** modifier requests that as well as outputting the substring that matched the entire pattern, pcretest should in addition output the remainder of the subject string. This is useful for tests where the subject contains multiple copies of the same substring.

The **/B** modifier is a debugging feature. It requests that **pcretest** output a representation of the compiled byte code after compilation.

The **/L** modifier must be followed directly by the name of a locale, for example,

```
 /pattern/Lfr_FR
```

For this reason, it must be the last modifier. The given locale is set, **pcre_maketables()** is called to build a set of character tables for the locale, and this is then passed to **pcre_compile()** when compiling the regular expression. Without an **/L** modifier, NULL is passed as the tables pointer; that is, **/L** applies only to the expression on which it appears.

The **/I** modifier requests that **pcretest** output information about the compiled pattern (whether it is anchored, has a fixed first character, and so on). It does this by calling **pcre_fullinfo()** after compiling a pattern. If the pattern is studied, the results of that are also output.

The **/D** modifier is a PCRE debugging feature, and is equivalent to **/BI**, that is, both the **/B** and the **/I** modifiers.

The **/F** modifier causes **pcretest** to flip the byte order of the fields in the compiled pattern that contain 2-byte and 4-byte numbers. This facility is for testing the feature in PCRE that allows it to execute patterns that were compiled on a host with a different endianness. This feature is not available when the POSIX interface to PCRE is being used, that is, when the **/P** pattern modifier is specified. See also the section about saving and reloading compiled patterns below.

The **/S** modifier causes **pcre_study()** to be called after the expression has been compiled, and the results used when the expression is matched.

The **/M** modifier causes the size of memory block used to hold the compiled pattern to be output.

The **/P** modifier causes **pcretest** to call PCRE via the POSIX wrapper API rather than its native API. When this is done, all other modifiers except **/i**, **/m**, and **/+** are ignored. REG_ICASE is set if **/i** is present, and REG_NEWLINE is set if **/m** is present. The wrapper functions force PCRE_DOLLAR_ENDONLY always, and PCRE_DOTALL unless REG_NEWLINE is set.

The **/8** modifier causes **pcretest** to call PCRE with the PCRE_UTF8 option set. This turns on support for UTF-8 character handling in PCRE, provided that it was compiled with this support enabled. This modifier also causes any non-printing characters in output strings to be printed using the \x{hh...} notation if they are valid UTF-8 sequences.

If the **/?** modifier is used with **/8**, it causes **pcretest** to call **pcre_compile()** with the PCRE_NO_UTF8_CHECK option, to suppress the checking of the string for UTF-8 validity.

DATA LINES

Before each data line is passed to **pcre_exec()**, leading and trailing whitespace is removed, and it is then scanned for \ escapes. Some of these are pretty esoteric features, intended for checking out some of the more complicated features of PCRE. If you are just testing "ordinary" regular expressions, you probably don't need any of these. The following escapes are recognized:

\a	alarm (BEL, \x07)
\b	backspace (\x08)
\e	escape (\x27)
\f	formfeed (\x0c)
\n	newline (\x0a)
\qdd	set the PCRE_MATCH_LIMIT limit to dd (any number of digits)
\r	carriage return (\x0d)
\t	tab (\x09)
\v	vertical tab (\x0b)
\nnn	octal character (up to 3 octal digits)
\xhh	hexadecimal character (up to 2 hex digits)
\x{hh...}	hexadecimal character, any number of digits in UTF-8 mode
\A	pass the PCRE_ANCHORED option to **pcre_exec()** or **pcre_dfa_exec()**
\B	pass the PCRE_NOTBOL option to **pcre_exec()** or **pcre_dfa_exec()**
\Cdd	call pcre_copy_substring() for substring dd after a successful match (number less than 32)
\Cname	call pcre_copy_named_substring() for substring "name" after a successful match (name terminated by next non alphanumeric character)
\C+	show the current captured substrings at callout time
\C-	do not supply a callout function
\C!n	return 1 instead of 0 when callout number n is reached
\C!n!m	return 1 instead of 0 when callout number n is reached for the nth time
\C*n	pass the number n (may be negative) as callout data; this is used as the callout return value
\D	use the **pcre_dfa_exec()** match function
\F	only shortest match for **pcre_dfa_exec()**
\Gdd	call pcre_get_substring() for substring dd after a successful match (number less than 32)
\Gname	call pcre_get_named_substring() for substring "name" after a successful match (name terminated by next non-alphanumeric character)
\L	call pcre_get_substringlist() after a successful match
\M	discover the minimum MATCH_LIMIT and MATCH_LIMIT_RECURSION settings
\N	pass the PCRE_NOTEMPTY option to **pcre_exec()** or **pcre_dfa_exec()**
\Odd	set the size of the output vector passed to

	pcre_exec() to dd (any number of digits)
\P	pass the PCRE_PARTIAL option to **pcre_exec()** or **pcre_dfa_exec()**
\Qdd	set the PCRE_MATCH_LIMIT_RECURSION limit to dd (any number of digits)
\R	pass the PCRE_DFA_RESTART option to **pcre_dfa_exec()**
\S	output details of memory get/free calls during matching
\Z	pass the PCRE_NOTEOL option to **pcre_exec()** or **pcre_dfa_exec()**
\?	pass the PCRE_NO_UTF8_CHECK option to **pcre_exec()** or **pcre_dfa_exec()**
\>dd	start the match at offset dd (any number of digits); this sets the *startoffset* argument for **pcre_exec()** or **pcre_dfa_exec()**
\<cr>	pass the PCRE_NEWLINE_CR option to **pcre_exec()** or **pcre_dfa_exec()**
\<lf>	pass the PCRE_NEWLINE_LF option to **pcre_exec()** or **pcre_dfa_exec()**
\<crlf>	pass the PCRE_NEWLINE_CRLF option to **pcre_exec()** or **pcre_dfa_exec()**
\<any>	pass the PCRE_NEWLINE_ANY option to **pcre_exec()** or **pcre_dfa_exec()**

The escapes that specify line ending sequences are literal strings, exactly as shown. No more than one newline setting should be present in any data line.

A backslash followed by anything else just escapes the anything else. If the very last character is a backslash, it is ignored. This gives a way of passing an empty line as data, since a real empty line terminates the data input.

If \M is present, **pcretest** calls **pcre_exec()** several times, with different values in the *match_limit* and *match_limit_recursion* fields of the **pcre_extra** data structure, until it finds the minimum numbers for each parameter that allow **pcre_exec()** to complete. The *match_limit* number is a measure of the amount of backtracking that takes place, and checking it out can be instructive. For most simple matches, the number is quite small, but for patterns with very large numbers of matching possibilities, it can become large very quickly with increasing length of subject string. The *match_limit_recursion* number is a measure of how much stack (or, if PCRE is compiled with NO_RECURSE, how much heap) memory is needed to complete the match attempt.

When \O is used, the value specified may be higher or lower than the size set by the **-O** command line option (or defaulted to 45); \O applies only to the call of **pcre_exec()** for the line in which it appears.

If the **/P** modifier was present on the pattern, causing the POSIX wrapper API to be used, the only option-setting sequences that have any effect are \B and \Z, causing REG_NOTBOL and REG_NOTEOL, respectively, to be passed to **regexec()**.

The use of \x{hh...} to represent UTF-8 characters is not dependent on the use of the **/8** modifier on the pattern. It is recognized always. There may be any number of hexadecimal digits inside the braces. The result is from one to six bytes, encoded according to the UTF-8 rules.

THE ALTERNATIVE MATCHING FUNCTION

By default, **pcretest** uses the standard PCRE matching function, **pcre_exec()** to match each data line. From release 6.0, PCRE supports an alternative matching function, **pcre_dfa_test()**, which operates in a different way, and has some restrictions. The differences between the two functions are described in the **pcrematching** documentation.

If a data line contains the \D escape sequence, or if the command line contains the **-dfa** option, the alternative matching function is called. This function finds all possible matches at a given point. If, however, the \F escape sequence is present in the data line, it stops after the first match is found. This is always the shortest possible match.

DEFAULT OUTPUT FROM PCRETEST

This section describes the output when the normal matching function, **pcre_exec()**, is being used.

When a match succeeds, pcretest outputs the list of captured substrings that **pcre_exec()** returns, starting with number 0 for the string that matched the whole pattern. Otherwise, it outputs "No match" or "Partial match" when **pcre_exec()** returns PCRE_ERROR_NOMATCH or PCRE_ERROR_PARTIAL, respectively, and otherwise the PCRE negative error number. Here is an example of an interactive **pcretest** run.

```
 $ pcretest
 PCRE version 7.0 30-Nov-2006

   re> /^abc(\d+)/
 data> abc123
  0: abc123
  1: 123
 data> xyz
 No match
```

If the strings contain any non-printing characters, they are output as \0x escapes, or as \x{...} escapes if the **/8** modifier was present on the pattern. See below for the definition of non-printing characters. If the pattern has the **/+** modifier, the output for substring 0 is followed by the the rest of the subject string, identified by "0+" like this:

```
   re> /cat/+
 data> cataract
  0: cat
  0+ aract
```

If the pattern has the **/g** or **/G** modifier, the results of successive matching attempts are output in sequence, like this:

```
   re> /\Bi(\w\w)/g
 data> Mississippi
  0: iss
  1: ss
  0: iss
  1: ss
  0: ipp
  1: pp
```

"No match" is output only if the first match attempt fails.

If any of the sequences **\C**, **\G**, or **\L** are present in a data line that is successfully matched, the substrings extracted by the convenience functions are output with C, G, or L after the string number instead of a colon. This is in addition to the normal full list. The string length (that is, the return from the extraction function) is given in parentheses after each string for **\C** and **\G**.

Note that whereas patterns can be continued over several lines (a plain ">" prompt is used for continuations), data lines may not. However newlines can be included in data by means of the \n escape (or \r, \r\n, etc., depending on the newline sequence setting).

OUTPUT FROM THE ALTERNATIVE MATCHING FUNCTION

When the alternative matching function, **pcre_dfa_exec()**, is used (by means of the \D escape sequence or the **-dfa** command line option), the output consists of a list of all the matches that start at the first point in the subject where there is at least one match. For example:

```
   re> /(tang|tangerine|tan)/
 data> yellow tangerine\D
```

```
  0: tangerine
  1: tang
  2: tan
```

(Using the normal matching function on this data finds only "tang".) The longest matching string is always given first (and numbered zero).

If **/g** is present on the pattern, the search for further matches resumes at the end of the longest match. For example:

```
    re> /(tang|tangerine|tan)/g
  data> yellow tangerine and tangy sultana\D
   0: tangerine
   1: tang
   2: tan
   0: tang
   1: tan
   0: tan
```

Since the matching function does not support substring capture, the escape sequences that are concerned with captured substrings are not relevant.

RESTARTING AFTER A PARTIAL MATCH

When the alternative matching function has given the PCRE_ERROR_PARTIAL return, indicating that the subject partially matched the pattern, you can restart the match with additional subject data by means of the \R escape sequence. For example:

```
    re> /^ ?(jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)$/
  data> 23ja\P\D
  Partial match: 23ja
  data> n05\R\D
   0: n05
```

For further information about partial matching, see the **pcrepartial** documentation.

CALLOUTS

If the pattern contains any callout requests, **pcretest**'s callout function is called during matching. This works with both matching functions. By default, the called function displays the callout number, the start and current positions in the text at the callout time, and the next pattern item to be tested. For example, the output

```
  --->pqrabcdef
    0    ^  ^     \d
```

indicates that callout number 0 occurred for a match attempt starting at the fourth character of the subject string, when the pointer was at the seventh character of the data, and when the next pattern item was \d. Just one circumflex is output if the start and current positions are the same.

Callouts numbered 255 are assumed to be automatic callouts, inserted as a result of the **/C** pattern modifier. In this case, instead of showing the callout number, the offset in the pattern, preceded by a plus, is output. For example:

```
    re> /\d?[A-E]\*/C
  data> E*
  --->E*
   +0 ^      \d?
   +3 ^      [A-E]
   +8 ^^     \*
  +10 ^ ^
  0: E*
```

The callout function in **pcretest** returns zero (carry on matching) by default, but you can use a \C item in a data line (as described above) to change this.

Inserting callouts can be helpful when using **pcretest** to check complicated regular expressions. For further information about callouts, see the **pcrecallout** documentation.

NON-PRINTING CHARACTERS

When **pcretest** is outputting text in the compiled version of a pattern, bytes other than 32-126 are always treated as non-printing characters are are therefore shown as hex escapes.

When **pcretest** is outputting text that is a matched part of a subject string, it behaves in the same way, unless a different locale has been set for the pattern (using the **/L** modifier). In this case, the **isprint()** function to distinguish printing and non-printing characters.

SAVING AND RELOADING COMPILED PATTERNS

The facilities described in this section are not available when the POSIX inteface to PCRE is being used, that is, when the **/P** pattern modifier is specified.

When the POSIX interface is not in use, you can cause **pcretest** to write a compiled pattern to a file, by following the modifiers with > and a file name. For example:

```
 /pattern/im >/some/file
```

See the **pcreprecompile** documentation for a discussion about saving and re-using compiled patterns.

The data that is written is binary. The first eight bytes are the length of the compiled pattern data followed by the length of the optional study data, each written as four bytes in big-endian order (most significant byte first). If there is no study data (either the pattern was not studied, or studying did not return any data), the second length is zero. The lengths are followed by an exact copy of the compiled pattern. If there is additional study data, this follows immediately after the compiled pattern. After writing the file, **pcretest** expects to read a new pattern.

A saved pattern can be reloaded into **pcretest** by specifing < and a file name instead of a pattern. The name of the file must not contain a < character, as otherwise **pcretest** will interpret the line as a pattern delimited by < characters. For example:

```
  re> </some/file
 Compiled regex loaded from /some/file
 No study data
```

When the pattern has been loaded, **pcretest** proceeds to read data lines in the usual way.

You can copy a file written by **pcretest** to a different host and reload it there, even if the new host has opposite endianness to the one on which the pattern was compiled. For example, you can compile on an i86 machine and run on a SPARC machine.

File names for saving and reloading can be absolute or relative, but note that the shell facility of expanding a file name that starts with a tilde (˜) is not available.

The ability to save and reload files in **pcretest** is intended for testing and experimentation. It is not intended for production use because only a single pattern can be written to a file. Furthermore, there is no facility for supplying custom character tables for use with a reloaded pattern. If the original pattern was compiled with custom tables, an attempt to match a subject string using a reloaded pattern is likely to cause **pcretest** to crash. Finally, if you attempt to load a file that is not in the correct format, the result is undefined.

SEE ALSO

pcre(3), **pcreapi**(3), **pcrecallout**(3), **pcrematching**(3), **pcrepartial**(d), **pcrepattern**(3), **pcreprecompile**(3).

AUTHOR

Philip Hazel
University Computing Service,
Cambridge CB2 3QH, England.

Last updated: 30 November 2006

NAME

PCRE - Perl-compatible regular expressions

INTRODUCTION

The PCRE library is a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl, with just a few differences. (Certain features that appeared in Python and PCRE before they appeared in Perl are also available using the Python syntax.)

The current implementation of PCRE (release 7.x) corresponds approximately with Perl 5.10, including support for UTF-8 encoded strings and Unicode general category properties. However, UTF-8 and Unicode support has to be explicitly enabled; it is not the default. The Unicode tables correspond to Unicode release 5.0.0.

In addition to the Perl-compatible matching function, PCRE contains an alternative matching function that matches the same compiled patterns in a different way. In certain circumstances, the alternative function has some advantages. For a discussion of the two matching algorithms, see the **pcrematching** page.

PCRE is written in C and released as a C library. A number of people have written wrappers and interfaces of various kinds. In particular, Google Inc. have provided a comprehensive C++ wrapper. This is now included as part of the PCRE distribution. The **pcrecpp** page has details of this interface. Other people's contributions can be found in the *Contrib* directory at the primary FTP site, which is:

ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre

Details of exactly which Perl regular expression features are and are not supported by PCRE are given in separate documents. See the **pcrepattern** and **pcrecompat** pages.

Some features of PCRE can be included, excluded, or changed when the library is built. The **pcre_config()** function makes it possible for a client to discover which features are available. The features themselves are described in the **pcrebuild** page. Documentation about building PCRE for various operating systems can be found in the **README** file in the source distribution.

The library contains a number of undocumented internal functions and data tables that are used by more than one of the exported external functions, but which are not intended for use by external callers. Their names all begin with "_pcre_", which hopefully will not provoke any name clashes. In some environments, it is possible to control which external symbols are exported when a shared library is built, and in these cases the undocumented symbols are not exported.

USER DOCUMENTATION

The user documentation for PCRE comprises a number of different sections. In the "man" format, each of these is a separate "man page". In the HTML format, each is a separate page, linked from the index page. In the plain text format, all the sections are concatenated, for ease of searching. The sections are as follows:

pcre	this document
pcreapi	details of PCRE's native C API
pcrebuild	options for building PCRE
pcrecallout	details of the callout feature
pcrecompat	discussion of Perl compatibility
pcrecpp	details of the C++ wrapper
pcregrep	description of the **pcregrep** command
pcrematching	discussion of the two matching algorithms
pcrepartial	details of the partial matching facility
pcrepattern	syntax and semantics of supported regular expressions
pcreperform	discussion of performance issues
pcreposix	the POSIX-compatible C API
pcreprecompile	details of saving and re-using precompiled patterns
pcresample	discussion of the sample program

pcrestack	discussion of stack usage
pcretest	description of the **pcretest** testing command

In addition, in the "man" and HTML formats, there is a short page for each C library function, listing its arguments and results.

LIMITATIONS

There are some size limitations in PCRE but it is hoped that they will never in practice be relevant.

The maximum length of a compiled pattern is 65539 (sic) bytes if PCRE is compiled with the default internal linkage size of 2. If you want to process regular expressions that are truly enormous, you can compile PCRE with an internal linkage size of 3 or 4 (see the **README** file in the source distribution and the **pcrebuild** documentation for details). In these cases the limit is substantially larger. However, the speed of execution is slower.

All values in repeating quantifiers must be less than 65536. The maximum compiled length of subpattern with an explicit repeat count is 30000 bytes. The maximum number of capturing subpatterns is 65535.

There is no limit to the number of parenthesized subpatterns, but there can be no more than 65535 capturing subpatterns.

The maximum length of name for a named subpattern is 32 characters, and the maximum number of named subpatterns is 10000.

The maximum length of a subject string is the largest positive number that an integer variable can hold. However, when using the traditional matching function, PCRE uses recursion to handle subpatterns and indefinite repetition. This means that the available stack space may limit the size of a subject string that can be processed by certain patterns. For a discussion of stack issues, see the **pcrestack** documentation.

UTF-8 AND UNICODE PROPERTY SUPPORT

From release 3.3, PCRE has had some support for character strings encoded in the UTF-8 format. For release 4.0 this was greatly extended to cover most common requirements, and in release 5.0 additional support for Unicode general category properties was added.

In order process UTF-8 strings, you must build PCRE to include UTF-8 support in the code, and, in addition, you must call **pcre_compile()** with the PCRE_UTF8 option flag. When you do this, both the pattern and any subject strings that are matched against it are treated as UTF-8 strings instead of just strings of bytes.

If you compile PCRE with UTF-8 support, but do not use it at run time, the library will be a bit bigger, but the additional run time overhead is limited to testing the PCRE_UTF8 flag occasionally, so should not be very big.

If PCRE is built with Unicode character property support (which implies UTF-8 support), the escape sequences \p{..}, \P{..}, and \X are supported. The available properties that can be tested are limited to the general category properties such as Lu for an upper case letter or Nd for a decimal number, the Unicode script names such as Arabic or Han, and the derived properties Any and L&. A full list is given in the **pcrepattern** documentation. Only the short names for properties are supported. For example, \p{L} matches a letter. Its Perl synonym, \p{Letter}, is not supported. Furthermore, in Perl, many properties may optionally be prefixed by "Is", for compatibility with Perl 5.6. PCRE does not support this.

The following comments apply when PCRE is running in UTF-8 mode:

1. When you set the PCRE_UTF8 flag, the strings passed as patterns and subjects are checked for validity on entry to the relevant functions. If an invalid UTF-8 string is passed, an error return is given. In some situations, you may already know that your strings are valid, and therefore want to skip these checks in order to improve performance. If you set the PCRE_NO_UTF8_CHECK flag at compile time or at run time, PCRE assumes that the pattern or subject it is given (respectively) contains only valid UTF-8 codes. In this case, it does not diagnose an invalid UTF-8 string. If you pass an invalid UTF-8 string to PCRE when PCRE_NO_UTF8_CHECK is set, the results are undefined. Your program may crash.

2. An unbraced hexadecimal escape sequence (such as \xb3) matches a two-byte UTF-8 character if the value is greater than 127.

3. Octal numbers up to \777 are recognized, and match two-byte UTF-8 characters for values greater than \177.

4. Repeat quantifiers apply to complete UTF-8 characters, not to individual bytes, for example: \x{100}{3}.

5. The dot metacharacter matches one UTF-8 character instead of a single byte.

6. The escape sequence \C can be used to match a single byte in UTF-8 mode, but its use can lead to some strange effects. This facility is not available in the alternative matching function, **pcre_dfa_exec()**.

7. The character escapes \b, \B, \d, \D, \s, \S, \w, and \W correctly test characters of any code value, but the characters that PCRE recognizes as digits, spaces, or word characters remain the same set as before, all with values less than 256. This remains true even when PCRE includes Unicode property support, because to do otherwise would slow down PCRE in many common cases. If you really want to test for a wider sense of, say, "digit", you must use Unicode property tests such as \p{Nd}.

8. Similarly, characters that match the POSIX named character classes are all low-valued characters.

9. Case-insensitive matching applies only to characters whose values are less than 128, unless PCRE is built with Unicode property support. Even when Unicode property support is available, PCRE still uses its own character tables when checking the case of low-valued characters, so as not to degrade performance. The Unicode property information is used only for characters with higher values. Even when Unicode property support is available, PCRE supports case-insensitive matching only when there is a one-to-one mapping between a letter’s cases. There are a small number of many-to-one mappings in Unicode; these are not supported by PCRE.

AUTHOR

Philip Hazel
University Computing Service,
Cambridge CB2 3QH, England.

Putting an actual email address here seems to have been a spam magnet, so I’ve taken it away. If you want to email me, use my initial and surname, separated by a dot, at the domain ucs.cam.ac.uk.

Last updated: 23 November 2006

NAME

PCRE - Perl-compatible regular expressions

PCRE NATIVE API

#include <pcre.h>

pcre *pcre_compile(const char ****pattern*, int** *options***,**
const char *****errptr*, int ****erroffset***,**
const unsigned char ****tableptr*);**

pcre *pcre_compile2(const char ****pattern*, int** *options***,**
int ****errorcodeptr*,**
const char *****errptr*, int ****erroffset***,**
const unsigned char ****tableptr*);**

pcre_extra *pcre_study(const pcre ****code*, int** *options***,**
const char *****errptr*);**

int pcre_exec(const pcre ****code*, const pcre_extra ****extra***,**
const char ****subject*, int** *length***, int** *startoffset***,**
int *options***, int ****ovector***, int** *ovecsize***);**

int pcre_dfa_exec(const pcre ****code*, const pcre_extra ****extra***,**
const char ****subject*, int** *length***, int** *startoffset***,**
int *options***, int ****ovector***, int** *ovecsize***,**
int ****workspace*, int** *wscount***);**

int pcre_copy_named_substring(const pcre ****code*,**
const char ****subject*, int ****ovector***,**
int *stringcount***, const char ****stringname***,**
char ****buffer*, int** *buffersize***);**

int pcre_copy_substring(const char ****subject*, int ****ovector***,**
int *stringcount***, int** *stringnumber***, char ****buffer***,**
int *buffersize***);**

int pcre_get_named_substring(const pcre ****code*,**
const char ****subject*, int ****ovector***,**
int *stringcount***, const char ****stringname***,**
const char *****stringptr*);**

int pcre_get_stringnumber(const pcre ****code*,**
const char ****name*);**

int pcre_get_stringtable_entries(const pcre ****code*,**
const char ****name*, char *****first***, char *****last***);**

int pcre_get_substring(const char ****subject*, int ****ovector***,**
int *stringcount***, int** *stringnumber***,**
const char *****stringptr*);**

int pcre_get_substring_list(const char ****subject*,**
int ****ovector*, int** *stringcount***, const char ******listptr***);**

void pcre_free_substring(const char ****stringptr*);**

void pcre_free_substring_list(const char *****stringptr*);**

const unsigned char *pcre_maketables(void);

int pcre_fullinfo(const pcre ****code*, const pcre_extra ****extra***,**
int *what***, void ****where***);**

int pcre_info(const pcre ****code*, int ****optptr***, int ****firstcharptr***);**

int pcre_refcount(pcre ****code*, int** *adjust***);**

int pcre_config(int *what***, void ****where***);**

char *pcre_version(void);

void *(*pcre_malloc)(size_t);

void (*pcre_free)(void *);

void *(*pcre_stack_malloc)(size_t);

void (*pcre_stack_free)(void *);

int (*pcre_callout)(pcre_callout_block *);

PCRE API OVERVIEW

PCRE has its own native API, which is described in this document. There are also some wrapper functions that correspond to the POSIX regular expression API. These are described in the **pcreposix** documentation. Both of these APIs define a set of C function calls. A C++ wrapper is distributed with PCRE. It is documented in the **pcrecpp** page.

The native API C function prototypes are defined in the header file **pcre.h**, and on Unix systems the library itself is called **libpcre**. It can normally be accessed by adding **-lpcre** to the command for linking an application that uses PCRE. The header file defines the macros PCRE_MAJOR and PCRE_MINOR to contain the major and minor release numbers for the library. Applications can use these to include support for different releases of PCRE.

The functions **pcre_compile()**, **pcre_compile2()**, **pcre_study()**, and **pcre_exec()** are used for compiling and matching regular expressions in a Perl-compatible manner. A sample program that demonstrates the simplest way of using them is provided in the file called *pcredemo.c* in the source distribution. The **pcresample** documentation describes how to run it.

A second matching function, **pcre_dfa_exec()**, which is not Perl-compatible, is also provided. This uses a different algorithm for the matching. The alternative algorithm finds all possible matches (at a given point in the subject), and scans the subject just once. However, this algorithm does not return captured substrings. A description of the two matching algorithms and their advantages and disadvantages is given in the **pcrematching** documentation.

In addition to the main compiling and matching functions, there are convenience functions for extracting captured substrings from a subject string that is matched by **pcre_exec()**. They are:

pcre_copy_substring()
pcre_copy_named_substring()
pcre_get_substring()
pcre_get_named_substring()
pcre_get_substring_list()
pcre_get_stringnumber()
pcre_get_stringtable_entries()

pcre_free_substring() and **pcre_free_substring_list()** are also provided, to free the memory used for extracted strings.

The function **pcre_maketables()** is used to build a set of character tables in the current locale for passing to **pcre_compile()**, **pcre_exec()**, or **pcre_dfa_exec()**. This is an optional facility that is provided for specialist use. Most commonly, no special tables are passed, in which case internal tables that are generated when PCRE is built are used.

The function **pcre_fullinfo()** is used to find out information about a compiled pattern; **pcre_info()** is an obsolete version that returns only some of the available information, but is retained for backwards compatibility. The function **pcre_version()** returns a pointer to a string containing the version of PCRE and its date of release.

The function **pcre_refcount()** maintains a reference count in a data block containing a compiled pattern. This is provided for the benefit of object-oriented applications.

The global variables **pcre_malloc** and **pcre_free** initially contain the entry points of the standard **malloc()** and **free()** functions, respectively. PCRE calls the memory management functions via these variables, so a calling program can replace them if it wishes to intercept the calls. This should be done before calling any PCRE functions.

The global variables **pcre_stack_malloc** and **pcre_stack_free** are also indirections to memory management functions. These special functions are used only when PCRE is compiled to use the heap for remembering data, instead of recursive function calls, when running the **pcre_exec()** function. See the **pcrebuild** documentation for details of how to do this. It is a non-standard way of building PCRE, for use in environments that have limited stacks. Because of the greater use of memory management, it runs more slowly. Separate functions are provided so that special-purpose external code can be used for this case. When used, these functions are always called in a stack-like manner (last obtained, first freed), and always for memory blocks of the same size. There is a discussion about PCRE's stack usage in the **pcrestack** documentation.

The global variable **pcre_callout** initially contains NULL. It can be set by the caller to a "callout" function, which PCRE will then call at specified points during a matching operation. Details are given in the **pcrecallout** documentation.

NEWLINES

PCRE supports four different conventions for indicating line breaks in strings: a single CR (carriage return) character, a single LF (linefeed) character, the two-character sequence CRLF, or any Unicode newline sequence. The Unicode newline sequences are the three just mentioned, plus the single characters VT (vertical tab, U+000B), FF (formfeed, U+000C), NEL (next line, U+0085), LS (line separator, U+2028), and PS (paragraph separator, U+2029).

Each of the first three conventions is used by at least one operating system as its standard newline sequence. When PCRE is built, a default can be specified. The default default is LF, which is the Unix standard. When PCRE is run, the default can be overridden, either when a pattern is compiled, or when it is matched.

In the PCRE documentation the word "newline" is used to mean "the character or pair of characters that indicate a line break". The choice of newline convention affects the handling of the dot, circumflex, and dollar metacharacters, the handling of #-comments in /x mode, and, when CRLF is a recognized line ending sequence, the match position advancement for a non-anchored pattern. The choice of newline convention does not affect the interpretation of the \n or \r escape sequences.

MULTITHREADING

The PCRE functions can be used in multi-threading applications, with the proviso that the memory management functions pointed to by **pcre_malloc**, **pcre_free**, **pcre_stack_malloc**, and **pcre_stack_free**, and the callout function pointed to by **pcre_callout**, are shared by all threads.

The compiled form of a regular expression is not altered during matching, so the same compiled pattern can safely be used by several threads at once.

SAVING PRECOMPILED PATTERNS FOR LATER USE

The compiled form of a regular expression can be saved and re-used at a later time, possibly by a different program, and even on a host other than the one on which it was compiled. Details are given in the **pcreprecompile** documentation.

CHECKING BUILD-TIME OPTIONS

int pcre_config(int *what*, **void** ***where);**

The function **pcre_config()** makes it possible for a PCRE client to discover which optional features have been compiled into the PCRE library. The **pcrebuild** documentation has more details about these optional features.

The first argument for **pcre_config()** is an integer, specifying which information is required; the second argument is a pointer to a variable into which the information is placed. The following information is available:

```
  PCRE_CONFIG_UTF8
```

The output is an integer that is set to one if UTF-8 support is available; otherwise it is set to zero.

```
  PCRE_CONFIG_UNICODE_PROPERTIES
```

The output is an integer that is set to one if support for Unicode character properties is available; otherwise it is set to zero.

```
  PCRE_CONFIG_NEWLINE
```

The output is an integer whose value specifies the default character sequence that is recognized as meaning "newline". The four values that are supported are: 10 for LF, 13 for CR, 3338 for CRLF, and -1 for ANY. The default should normally be the standard sequence for your operating system.

```
  PCRE_CONFIG_LINK_SIZE
```

The output is an integer that contains the number of bytes used for internal linkage in compiled regular expressions. The value is 2, 3, or 4. Larger values allow larger regular expressions to be compiled, at the expense of slower matching. The default value of 2 is sufficient for all but the most massive patterns, since it allows the compiled pattern to be up to 64K in size.

```
  PCRE_CONFIG_POSIX_MALLOC_THRESHOLD
```

The output is an integer that contains the threshold above which the POSIX interface uses **malloc()** for output vectors. Further details are given in the **pcreposix** documentation.

```
  PCRE_CONFIG_MATCH_LIMIT
```

The output is an integer that gives the default limit for the number of internal matching function calls in a **pcre_exec()** execution. Further details are given with **pcre_exec()** below.

```
  PCRE_CONFIG_MATCH_LIMIT_RECURSION
```

The output is an integer that gives the default limit for the depth of recursion when calling the internal matching function in a **pcre_exec()** execution. Further details are given with **pcre_exec()** below.

```
  PCRE_CONFIG_STACKRECURSE
```

The output is an integer that is set to one if internal recursion when running **pcre_exec()** is implemented by recursive function calls that use the stack to remember their state. This is the usual way that PCRE is compiled. The output is zero if PCRE was compiled to use blocks of data on the heap instead of recursive function calls. In this case, **pcre_stack_malloc** and **pcre_stack_free** are called to manage memory blocks on the heap, thus avoiding the use of the stack.

COMPILING A PATTERN

pcre *pcre_compile(const char ****pattern*, int** *options***,**
    const char *****errptr*, int ****erroffset***,**
    const unsigned char ****tableptr*);**

pcre *pcre_compile2(const char ****pattern*, int** *options***,**
    int ****errorcodeptr*,**
    const char *****errptr*, int ****erroffset***,**
    const unsigned char ****tableptr*);**

Either of the functions **pcre_compile()** or **pcre_compile2()** can be called to compile a pattern into an internal form. The only difference between the two interfaces is that **pcre_compile2()** has an additional argument, *errorcodeptr,* via which a numerical error code can be returned.

The pattern is a C string terminated by a binary zero, and is passed in the *pattern* argument. A pointer to a single block of memory that is obtained via **pcre_malloc** is returned. This contains the compiled code and related data. The **pcre** type is defined for the returned block; this is a typedef for a structure whose contents are not externally defined. It is up to the caller to free the memory (via **pcre_free**) when it is no longer required.

Although the compiled code of a PCRE regex is relocatable, that is, it does not depend on memory location, the complete **pcre** data block is not fully relocatable, because it may contain a copy of the *tableptr* argument, which is an address (see below).

The *options* argument contains various bit settings that affect the compilation. It should be zero if no options are required. The available options are described below. Some of them, in particular, those that are compatible with Perl, can also be set and unset from within the pattern (see the detailed description in the **pcrepattern** documentation). For these options, the contents of the *options* argument specifies their initial settings at the start of compilation and execution. The PCRE_ANCHORED and PCRE_NEWLINE_*xxx* options can be set at the time of matching as well as at compile time.

If *errptr* is NULL, **pcre_compile()** returns NULL immediately. Otherwise, if compilation of a pattern fails, **pcre_compile()** returns NULL, and sets the variable pointed to by *errptr* to point to a textual error message. This is a static string that is part of the library. You must not try to free it. The offset from the start of the pattern to the character where the error was discovered is placed in the variable pointed to by *erroffset*, which must not be NULL. If it is, an immediate error is given.

If **pcre_compile2()** is used instead of **pcre_compile()**, and the *errorcodeptr* argument is not NULL, a non-zero error code number is returned via this argument in the event of an error. This is in addition to the textual error message. Error codes and messages are listed below.

If the final argument, *tableptr*, is NULL, PCRE uses a default set of character tables that are built when PCRE is compiled, using the default C locale. Otherwise, *tableptr* must be an address that is the result of a call to **pcre_maketables()**. This value is stored with the compiled pattern, and used again by **pcre_exec()**, unless another table pointer is passed to it. For more discussion, see the section on locale support below.

This code fragment shows a typical straightforward call to **pcre_compile()**:

```
pcre *re;
const char *error;
int erroffset;
re = pcre_compile(
  "^A.*Z",          /* the pattern */
  0,                /* default options */
  &error,           /* for error message */
  &erroffset,       /* for error offset */
  NULL);            /* use default character tables */
```

The following names for option bits are defined in the **pcre.h** header file:

```
PCRE_ANCHORED
```

If this bit is set, the pattern is forced to be "anchored", that is, it is constrained to match only at the first matching point in the string that is being searched (the "subject string"). This effect can also be achieved by appropriate constructs in the pattern itself, which is the only way to do it in Perl.

```
PCRE_AUTO_CALLOUT
```

If this bit is set, **pcre_compile()** automatically inserts callout items, all with number 255, before each pattern item. For discussion of the callout facility, see the **pcrecallout** documentation.

```
PCRE_CASELESS
```

If this bit is set, letters in the pattern match both upper and lower case letters. It is equivalent to Perl's /i option, and it can be changed within a pattern by a (?i) option setting. In UTF-8 mode, PCRE always understands the concept of case for characters whose values are less than 128, so caseless matching is always possible. For characters with higher values, the concept of case is supported if PCRE is compiled with Unicode property support, but not otherwise. If you want to use caseless matching for characters 128 and above, you must ensure that PCRE is compiled with Unicode property support as well as with UTF-8 support.

```
PCRE_DOLLAR_ENDONLY
```

If this bit is set, a dollar metacharacter in the pattern matches only at the end of the subject string. Without this option, a dollar also matches immediately before a newline at the end of the string (but not before any other newlines). The PCRE_DOLLAR_ENDONLY option is ignored if PCRE_MULTILINE is set. There is no equivalent to this option in Perl, and no way to set it within a pattern.

```
  PCRE_DOTALL
```

If this bit is set, a dot metacharater in the pattern matches all characters, including those that indicate newline. Without it, a dot does not match when the current position is at a newline. This option is equivalent to Perl's /s option, and it can be changed within a pattern by a (?s) option setting. A negative class such as [ˆa] always matches newline characters, independent of the setting of this option.

```
  PCRE_DUPNAMES
```

If this bit is set, names used to identify capturing subpatterns need not be unique. This can be helpful for certain types of pattern when it is known that only one instance of the named subpattern can ever be matched. There are more details of named subpatterns below; see also the **pcrepattern** documentation.

```
  PCRE_EXTENDED
```

If this bit is set, whitespace data characters in the pattern are totally ignored except when escaped or inside a character class. Whitespace does not include the VT character (code 11). In addition, characters between an unescaped # outside a character class and the next newline, inclusive, are also ignored. This is equivalent to Perl's /x option, and it can be changed within a pattern by a (?x) option setting.

This option makes it possible to include comments inside complicated patterns. Note, however, that this applies only to data characters. Whitespace characters may never appear within special character sequences in a pattern, for example within the sequence (?(which introduces a conditional subpattern.

```
  PCRE_EXTRA
```

This option was invented in order to turn on additional functionality of PCRE that is incompatible with Perl, but it is currently of very little use. When set, any backslash in a pattern that is followed by a letter that has no special meaning causes an error, thus reserving these combinations for future expansion. By default, as in Perl, a backslash followed by a letter with no special meaning is treated as a literal. (Perl can, however, be persuaded to give a warning for this.) There are at present no other features controlled by this option. It can also be set by a (?X) option setting within a pattern.

```
  PCRE_FIRSTLINE
```

If this option is set, an unanchored pattern is required to match before or at the first newline in the subject string, though the matched text may continue over the newline.

```
  PCRE_MULTILINE
```

By default, PCRE treats the subject string as consisting of a single line of characters (even if it actually contains newlines). The "start of line" metacharacter (ˆ) matches only at the start of the string, while the "end of line" metacharacter ($) matches only at the end of the string, or before a terminating newline (unless PCRE_DOLLAR_ENDONLY is set). This is the same as Perl.

When PCRE_MULTILINE it is set, the "start of line" and "end of line" constructs match immediately following or immediately before internal newlines in the subject string, respectively, as well as at the very start and end. This is equivalent to Perl's /m option, and it can be changed within a pattern by a (?m) option setting. If there are no newlines in a subject string, or no occurrences of ˆ or $ in a pattern, setting PCRE_MULTILINE has no effect.

```
  PCRE_NEWLINE_CR
  PCRE_NEWLINE_LF
  PCRE_NEWLINE_CRLF
  PCRE_NEWLINE_ANY
```

These options override the default newline definition that was chosen when PCRE was built. Setting the first or the second specifies that a newline is indicated by a single character (CR or LF, respectively). Setting PCRE_NEWLINE_CRLF specifies that a newline is indicated by the two-character CRLF sequence. Setting PCRE_NEWLINE_ANY specifies that any Unicode newline sequence should be recognized. The Unicode newline sequences are the three just mentioned, plus the single characters VT (vertical tab, U+000B), FF (formfeed, U+000C), NEL (next line, U+0085), LS (line separator, U+2028), and PS (paragraph separator, U+2029). The last two are recognized only in UTF-8 mode.

The newline setting in the options word uses three bits that are treated as a number, giving eight possibilities. Currently only five are used (default plus the four values above). This means that if you set more than one newline option, the combination may or may not be sensible. For example, PCRE_NEWLINE_CR with PCRE_NEWLINE_LF is equivalent to PCRE_NEWLINE_CRLF, but other combinations yield unused numbers and cause an error.

The only time that a line break is specially recognized when compiling a pattern is if PCRE_EXTENDED is set, and an unescaped # outside a character class is encountered. This indicates a comment that lasts until after the next line break sequence. In other circumstances, line break sequences are treated as literal data, except that in PCRE_EXTENDED mode, both CR and LF are treated as whitespace characters and are therefore ignored.

The newline option that is set at compile time becomes the default that is used for **pcre_exec()** and **pcre_dfa_exec()**, but it can be overridden.

```
  PCRE_NO_AUTO_CAPTURE
```

If this option is set, it disables the use of numbered capturing parentheses in the pattern. Any opening parenthesis that is not followed by ? behaves as if it were followed by ?: but named parentheses can still be used for capturing (and they acquire numbers in the usual way). There is no equivalent of this option in Perl.

```
  PCRE_UNGREEDY
```

This option inverts the "greediness" of the quantifiers so that they are not greedy by default, but become greedy if followed by "?". It is not compatible with Perl. It can also be set by a (?U) option setting within the pattern.

```
  PCRE_UTF8
```

This option causes PCRE to regard both the pattern and the subject as strings of UTF-8 characters instead of single-byte character strings. However, it is available only when PCRE is built to include UTF-8 support. If not, the use of this option provokes an error. Details of how this option changes the behaviour of PCRE are given in the section on UTF-8 support in the main **pcre** page.

```
  PCRE_NO_UTF8_CHECK
```

When PCRE_UTF8 is set, the validity of the pattern as a UTF-8 string is automatically checked. If an invalid UTF-8 sequence of bytes is found, **pcre_compile()** returns an error. If you already know that your pattern is valid, and you want to skip this check for performance reasons, you can set the PCRE_NO_UTF8_CHECK option. When it is set, the effect of passing an invalid UTF-8 string as a pattern is undefined. It may cause your program to crash. Note that this option can also be passed to **pcre_exec()** and **pcre_dfa_exec()**, to suppress the UTF-8 validity checking of subject strings.

COMPILATION ERROR CODES

The following table lists the error codes than may be returned by **pcre_compile2()**, along with the error messages that may be returned by both compiling functions. As PCRE has developed, some error codes have fallen out of use. To avoid confusion, they have not been re-used.

```
   0  no error
   1  \ at end of pattern
   2  \c at end of pattern
   3  unrecognized character follows \
   4  numbers out of order in {} quantifier
   5  number too big in {} quantifier
   6  missing terminating ] for character class
   7  invalid escape sequence in character class
   8  range out of order in character class
   9  nothing to repeat
  10  [this code is not in use]
  11  internal error: unexpected repeat
  12  unrecognized character after (?
  13  POSIX named classes are supported only within a class
  14  missing )
  15  reference to non-existent subpattern
  16  erroffset passed as NULL
  17  unknown option bit(s) set
  18  missing ) after comment
  19  [this code is not in use]
  20  regular expression too large
  21  failed to get memory
  22  unmatched parentheses
  23  internal error: code overflow
  24  unrecognized character after (?<
  25  lookbehind assertion is not fixed length
  26  malformed number or name after (?(
  27  conditional group contains more than two branches
  28  assertion expected after (?(
  29  (?R or (?digits must be followed by )
  30  unknown POSIX class name
  31  POSIX collating elements are not supported
  32  this version of PCRE is not compiled with PCRE_UTF8 support
  33  [this code is not in use]
  34  character value in \x{...} sequence is too large
  35  invalid condition (?(0)
  36  \C not allowed in lookbehind assertion
  37  PCRE does not support \L, \l, \N, \U, or \u
  38  number after (?C is > 255
  39  closing ) for (?C expected
  40  recursive call could loop indefinitely
  41  unrecognized character after (?P
  42  syntax error in subpattern name (missing terminator)
  43  two named subpatterns have the same name
  44  invalid UTF-8 string
  45  support for \P, \p, and \X has not been compiled
  46  malformed \P or \p sequence
  47  unknown property name after \P or \p
  48  subpattern name is too long (maximum 32 characters)
  49  too many named subpatterns (maximum 10,000)
  50  repeated subpattern is too long
  51  octal value is greater than \377 (not in UTF-8 mode)
  52  internal error: overran compiling workspace
  53  internal error: previously-checked referenced subpattern not found
  54  DEFINE group contains more than one branch
  55  repeating a DEFINE group is not allowed
  56  inconsistent NEWLINE options"
```

STUDYING A PATTERN

pcre_extra *pcre_study(const pcre ****code*, **int *options*
const char *****errptr*);**

If a compiled pattern is going to be used several times, it is worth spending more time analyzing it in order to speed up the time taken for matching. The function **pcre_study()** takes a pointer to a compiled pattern as its first argument. If studying the pattern produces additional information that will help speed up matching, **pcre_study()** returns a pointer to a **pcre_extra** block, in which the *study_data* field points to the results of the study.

The returned value from **pcre_study()** can be passed directly to **pcre_exec()**. However, a **pcre_extra** block also contains other fields that can be set by the caller before the block is passed; these are described below in the section on matching a pattern.

If studying the pattern does not produce any additional information **pcre_study()** returns NULL. In that circumstance, if the calling program wants to pass any of the other fields to **pcre_exec()**, it must set up its own **pcre_extra** block.

The second argument of **pcre_study()** contains option bits. At present, no options are defined, and this argument should always be zero.

The third argument for **pcre_study()** is a pointer for an error message. If studying succeeds (even if no data is returned), the variable it points to is set to NULL. Otherwise it is set to point to a textual error message. This is a static string that is part of the library. You must not try to free it. You should test the error pointer for NULL after calling **pcre_study()**, to be sure that it has run successfully.

This is a typical call to **pcre_study**():

```
pcre_extra *pe;
pe = pcre_study(
  re,           /* result of pcre_compile() */
  0,            /* no options exist */
  &error);      /* set to NULL or points to a message */
```

At present, studying a pattern is useful only for non-anchored patterns that do not have a single fixed starting character. A bitmap of possible starting bytes is created.

LOCALE SUPPORT

PCRE handles caseless matching, and determines whether characters are letters digits, or whatever, by reference to a set of tables, indexed by character value. When running in UTF-8 mode, this applies only to characters with codes less than 128. Higher-valued codes never match escapes such as \w or \d, but can be tested with \p if PCRE is built with Unicode character property support. The use of locales with Unicode is discouraged.

An internal set of tables is created in the default C locale when PCRE is built. This is used when the final argument of **pcre_compile()** is NULL, and is sufficient for many applications. An alternative set of tables can, however, be supplied. These may be created in a different locale from the default. As more and more applications change to using Unicode, the need for this locale support is expected to die away.

External tables are built by calling the **pcre_maketables()** function, which has no arguments, in the relevant locale. The result can then be passed to **pcre_compile()** or **pcre_exec()** as often as necessary. For example, to build and use tables that are appropriate for the French locale (where accented characters with values greater than 128 are treated as letters), the following code could be used:

```
setlocale(LC_CTYPE, "fr_FR");
tables = pcre_maketables();
re = pcre_compile(..., tables);
```

When **pcre_maketables()** runs, the tables are built in memory that is obtained via **pcre_malloc**. It is the caller’s responsibility to ensure that the memory containing the tables remains available for as long as it is needed.

The pointer that is passed to **pcre_compile()** is saved with the compiled pattern, and the same tables are used via this pointer by **pcre_study()** and normally also by **pcre_exec()**. Thus, by default, for any single pattern, compilation, studying and matching all happen in the same locale, but different patterns can be compiled in different locales.

It is possible to pass a table pointer or NULL (indicating the use of the internal tables) to **pcre_exec()**. Although not intended for this purpose, this facility could be used to match a pattern in a different locale from the one in which it was compiled. Passing table pointers at run time is discussed below in the section on matching a pattern.

INFORMATION ABOUT A PATTERN

int pcre_fullinfo(const pcre ***_code_, const pcre_extra ***_extra_**,**
int _what_**, void ***_where_**);**

The **pcre_fullinfo()** function returns information about a compiled pattern. It replaces the obsolete **pcre_info()** function, which is nevertheless retained for backwards compability (and is documented below).

The first argument for **pcre_fullinfo()** is a pointer to the compiled pattern. The second argument is the result of **pcre_study()**, or NULL if the pattern was not studied. The third argument specifies which piece of information is required, and the fourth argument is a pointer to a variable to receive the data. The yield of the function is zero for success, or one of the following negative numbers:

```
PCRE_ERROR_NULL       the argument code was NULL
                      the argument where was NULL
PCRE_ERROR_BADMAGIC   the "magic number" was not found
PCRE_ERROR_BADOPTION  the value of what was invalid
```

The "magic number" is placed at the start of each compiled pattern as an simple check against passing an arbitrary memory pointer. Here is a typical call of **pcre_fullinfo()**, to obtain the length of the compiled pattern:

```
int rc;
size_t length;
rc = pcre_fullinfo(
  re,               /* result of pcre_compile() */
  pe,               /* result of pcre_study(), or NULL */
  PCRE_INFO_SIZE,   /* what is required */
  &length);         /* where to put the data */
```

The possible values for the third argument are defined in **pcre.h**, and are as follows:

```
PCRE_INFO_BACKREFMAX
```

Return the number of the highest back reference in the pattern. The fourth argument should point to an **int** variable. Zero is returned if there are no back references.

```
PCRE_INFO_CAPTURECOUNT
```

Return the number of capturing subpatterns in the pattern. The fourth argument should point to an **int** variable.

```
PCRE_INFO_DEFAULT_TABLES
```

Return a pointer to the internal default character tables within PCRE. The fourth argument should point to an **unsigned char *** variable. This information call is provided for internal use by the **pcre_study()** function. External callers can cause PCRE to use its internal tables by passing a NULL table pointer.

```
PCRE_INFO_FIRSTBYTE
```

Return information about the first byte of any matched string, for a non-anchored pattern. The fourth argument should point to an **int** variable. (This option used to be called PCRE_INFO_FIRSTCHAR; the old name is still recognized for backwards compatibility.)

If there is a fixed first byte, for example, from a pattern such as (cat|cow|coyote), its value is returned.

Otherwise, if either

(a) the pattern was compiled with the PCRE_MULTILINE option, and every branch starts with "^", or

(b) every branch of the pattern starts with ".*" and PCRE_DOTALL is not set (if it were set, the pattern would be anchored),

-1 is returned, indicating that the pattern matches only at the start of a subject string or after any newline within the string. Otherwise -2 is returned. For anchored patterns, -2 is returned.

```
  PCRE_INFO_FIRSTTABLE
```

If the pattern was studied, and this resulted in the construction of a 256-bit table indicating a fixed set of bytes for the first byte in any matching string, a pointer to the table is returned. Otherwise NULL is returned. The fourth argument should point to an **unsigned char *** variable.

```
  PCRE_INFO_LASTLITERAL
```

Return the value of the rightmost literal byte that must exist in any matched string, other than at its start, if such a byte has been recorded. The fourth argument should point to an **int** variable. If there is no such byte, -1 is returned. For anchored patterns, a last literal byte is recorded only if it follows something of variable length. For example, for the pattern /^a\d+z\d+/ the returned value is "z", but for /^a\dz\d/ the returned value is -1.

```
  PCRE_INFO_NAMECOUNT
  PCRE_INFO_NAMEENTRYSIZE
  PCRE_INFO_NAMETABLE
```

PCRE supports the use of named as well as numbered capturing parentheses. The names are just an additional way of identifying the parentheses, which still acquire numbers. Several convenience functions such as **pcre_get_named_substring()** are provided for extracting captured substrings by name. It is also possible to extract the data directly, by first converting the name to a number in order to access the correct pointers in the output vector (described with **pcre_exec()** below). To do the conversion, you need to use the name-to-number map, which is described by these three values.

The map consists of a number of fixed-size entries. PCRE_INFO_NAMECOUNT gives the number of entries, and PCRE_INFO_NAMEENTRYSIZE gives the size of each entry; both of these return an **int** value. The entry size depends on the length of the longest name. PCRE_INFO_NAMETABLE returns a pointer to the first entry of the table (a pointer to **char**). The first two bytes of each entry are the number of the capturing parenthesis, most significant byte first. The rest of the entry is the corresponding name, zero terminated. The names are in alphabetical order. When PCRE_DUPNAMES is set, duplicate names are in order of their parentheses numbers. For example, consider the following pattern (assume PCRE_EXTENDED is set, so white space - including newlines - is ignored):

```
  (?<date> (?<year>(\d\d)?\d\d) -
  (?<month>\d\d) - (?<day>\d\d) )
```

There are four named subpatterns, so the table has four entries, and each entry in the table is eight bytes long. The table is as follows, with non-printing bytes shows in hexadecimal, and undefined bytes shown as ??:

```
  00 01 d  a  t  e  00 ??
  00 05 d  a  y  00 ?? ??
  00 04 m  o  n  t  h  00
  00 02 y  e  a  r  00 ??
```

When writing code to extract data from named subpatterns using the name-to-number map, remember that the length of the entries is likely to be different for each compiled pattern.

```
  PCRE_INFO_OPTIONS
```

Return a copy of the options with which the pattern was compiled. The fourth argument should point to an **unsigned long int** variable. These option bits are those specified in the call to **pcre_compile()**, modified by any top-level option settings within the pattern itself.

A pattern is automatically anchored by PCRE if all of its top-level alternatives begin with one of the following:

```
  ^    unless PCRE_MULTILINE is set
  \A   always
  \G   always
  .*   if PCRE_DOTALL is set and there are no back
         references to the subpattern in which .* appears
```

For such patterns, the PCRE_ANCHORED bit is set in the options returned by **pcre_fullinfo()**.

```
  PCRE_INFO_SIZE
```

Return the size of the compiled pattern, that is, the value that was passed as the argument to **pcre_malloc()** when PCRE was getting memory in which to place the compiled data. The fourth argument should point to a **size_t** variable.

```
  PCRE_INFO_STUDYSIZE
```

Return the size of the data block pointed to by the *study_data* field in a **pcre_extra** block. That is, it is the value that was passed to **pcre_malloc()** when PCRE was getting memory into which to place the data created by **pcre_study()**. The fourth argument should point to a **size_t** variable.

OBSOLETE INFO FUNCTION

int pcre_info(const pcre ****code*, int ****optptr***, int ****firstcharptr***);**

The **pcre_info()** function is now obsolete because its interface is too restrictive to return all the available data about a compiled pattern. New programs should use **pcre_fullinfo()** instead. The yield of **pcre_info()** is the number of capturing subpatterns, or one of the following negative numbers:

```
  PCRE_ERROR_NULL       the argument code was NULL
  PCRE_ERROR_BADMAGIC   the "magic number" was not found
```

If the *optptr* argument is not NULL, a copy of the options with which the pattern was compiled is placed in the integer it points to (see PCRE_INFO_OPTIONS above).

If the pattern is not anchored and the *firstcharptr* argument is not NULL, it is used to pass back information about the first character of any matched string (see PCRE_INFO_FIRSTBYTE above).

REFERENCE COUNTS

int pcre_refcount(pcre ****code*, int** *adjust***);**

The **pcre_refcount()** function is used to maintain a reference count in the data block that contains a compiled pattern. It is provided for the benefit of applications that operate in an object-oriented manner, where different parts of the application may be using the same compiled pattern, but you want to free the block when they are all done.

When a pattern is compiled, the reference count field is initialized to zero. It is changed only by calling this function, whose action is to add the *adjust* value (which may be positive or negative) to it. The yield of the function is the new value. However, the value of the count is constrained to lie between 0 and 65535, inclusive. If the new value is outside these limits, it is forced to the appropriate limit value.

Except when it is zero, the reference count is not correctly preserved if a pattern is compiled on one host and then transferred to a host whose byte-order is different. (This seems a highly unlikely scenario.)

MATCHING A PATTERN: THE TRADITIONAL FUNCTION

int pcre_exec(const pcre ****code*, const pcre_extra ****extra***,**
const char ****subject*, int** *length***, int** *startoffset***,**
int *options***, int ****ovector***, int** *ovecsize***);**

The function **pcre_exec()** is called to match a subject string against a compiled pattern, which is passed in the *code* argument. If the pattern has been studied, the result of the study should be passed in the *extra* argument. This function is the main matching facility of the library, and it operates in a Perl-like manner. For specialist use there is also an alternative matching function, which is described below in the section about the **pcre_dfa_exec()** function.

In most applications, the pattern will have been compiled (and optionally studied) in the same process that calls **pcre_exec()**. However, it is possible to save compiled patterns and study data, and then use them later in different processes, possibly even on different hosts. For a discussion about this, see the **pcreprecompile** documentation.

Here is an example of a simple call to **pcre_exec()**:

```
int rc;
int ovector[30];
rc = pcre_exec(
  re,             /* result of pcre_compile() */
  NULL,           /* we didn't study the pattern */
  "some string",  /* the subject string */
  11,             /* the length of the subject string */
  0,              /* start at offset 0 in the subject */
  0,              /* default options */
  ovector,        /* vector of integers for substring information */
  30);            /* number of elements (NOT size in bytes) */
```

Extra data for pcre_exec()

If the *extra* argument is not NULL, it must point to a **pcre_extra** data block. The **pcre_study()** function returns such a block (when it doesn't return NULL), but you can also create one for yourself, and pass additional information in it. The **pcre_extra** block contains the following fields (not necessarily in this order):

```
unsigned long int flags;
void *study_data;
unsigned long int match_limit;
unsigned long int match_limit_recursion;
void *callout_data;
const unsigned char *tables;
```

The *flags* field is a bitmap that specifies which of the other fields are set. The flag bits are:

```
PCRE_EXTRA_STUDY_DATA
PCRE_EXTRA_MATCH_LIMIT
PCRE_EXTRA_MATCH_LIMIT_RECURSION
PCRE_EXTRA_CALLOUT_DATA
PCRE_EXTRA_TABLES
```

Other flag bits should be set to zero. The *study_data* field is set in the **pcre_extra** block that is returned by **pcre_study()**, together with the appropriate flag bit. You should not set this yourself, but you may add to the block by setting the other fields and their corresponding flag bits.

The *match_limit* field provides a means of preventing PCRE from using up a vast amount of resources when running patterns that are not going to match, but which have a very large number of possibilities in their search trees. The classic example is the use of nested unlimited repeats.

Internally, PCRE uses a function called **match()** which it calls repeatedly (sometimes recursively). The limit set by *match_limit* is imposed on the number of times this function is called during a match, which has the effect of limiting the amount of backtracking that can take place. For patterns that are not anchored, the count restarts from zero for each position in the subject string.

The default value for the limit can be set when PCRE is built; the default default is 10 million, which handles all but the most extreme cases. You can override the default by suppling **pcre_exec()** with a **pcre_extra** block in which *match_limit* is set, and PCRE_EXTRA_MATCH_LIMIT is set in the *flags* field. If the limit is exceeded, **pcre_exec()** returns PCRE_ERROR_MATCHLIMIT.

The *match_limit_recursion* field is similar to *match_limit*, but instead of limiting the total number of times that **match()** is called, it limits the depth of recursion. The recursion depth is a smaller number than the total number of calls, because not all calls to **match()** are recursive. This limit is of use only if it is set smaller than *match_limit*.

Limiting the recursion depth limits the amount of stack that can be used, or, when PCRE has been compiled to use memory on the heap instead of the stack, the amount of heap memory that can be used.

The default value for *match_limit_recursion* can be set when PCRE is built; the default default is the same value as the default for *match_limit*. You can override the default by suppling **pcre_exec()** with a **pcre_extra** block in which *match_limit_recursion* is set, and PCRE_EXTRA_MATCH_LIMIT_RECURSION is set in the *flags* field. If the limit is exceeded, **pcre_exec()** returns PCRE_ERROR_RECURSIONLIMIT.

The *pcre_callout* field is used in conjunction with the "callout" feature, which is described in the **pcrecallout** documentation.

The *tables* field is used to pass a character tables pointer to **pcre_exec()**; this overrides the value that is stored with the compiled pattern. A non-NULL value is stored with the compiled pattern only if custom tables were supplied to **pcre_compile()** via its *tableptr* argument. If NULL is passed to **pcre_exec()** using this mechanism, it forces PCRE's internal tables to be used. This facility is helpful when re-using patterns that have been saved after compiling with an external set of tables, because the external tables might be at a different address when **pcre_exec()** is called. See the **pcreprecompile** documentation for a discussion of saving compiled patterns for later use.

Option bits for pcre_exec()

The unused bits of the *options* argument for **pcre_exec()** must be zero. The only bits that may be set are PCRE_ANCHORED, PCRE_NEWLINE_*xxx*, PCRE_NOTBOL, PCRE_NOTEOL, PCRE_NOTEMPTY, PCRE_NO_UTF8_CHECK and PCRE_PARTIAL.

```
  PCRE_ANCHORED
```

The PCRE_ANCHORED option limits **pcre_exec()** to matching at the first matching position. If a pattern was compiled with PCRE_ANCHORED, or turned out to be anchored by virtue of its contents, it cannot be made unachored at matching time.

```
  PCRE_NEWLINE_CR
  PCRE_NEWLINE_LF
  PCRE_NEWLINE_CRLF
  PCRE_NEWLINE_ANY
```

These options override the newline definition that was chosen or defaulted when the pattern was compiled. For details, see the description of **pcre_compile()** above. During matching, the newline choice affects the behaviour of the dot, circumflex, and dollar metacharacters. It may also alter the way the match position is advanced after a match failure for an unanchored pattern. When PCRE_NEWLINE_CRLF or PCRE_NEWLINE_ANY is set, and a match attempt fails when the current position is at a CRLF sequence, the match position is advanced by two characters instead of one, in other words, to after the CRLF.

```
  PCRE_NOTBOL
```

This option specifies that first character of the subject string is not the beginning of a line, so the circumflex metacharacter should not match before it. Setting this without PCRE_MULTILINE (at compile time) causes circumflex never to match. This option affects only the behaviour of the circumflex metacharacter. It does not affect \A.

```
  PCRE_NOTEOL
```

This option specifies that the end of the subject string is not the end of a line, so the dollar metacharacter should not match it nor (except in multiline mode) a newline immediately before it. Setting this without PCRE_MULTILINE (at compile time) causes dollar never to match. This option affects only the behaviour of the dollar metacharacter. It does not affect \Z or \z.

```
  PCRE_NOTEMPTY
```

An empty string is not considered to be a valid match if this option is set. If there are alternatives in the pattern, they are tried. If all the alternatives match the empty string, the entire match fails. For example, if the pattern

```
  a?b?
```

is applied to a string not beginning with "a" or "b", it matches the empty string at the start of the subject. With PCRE_NOTEMPTY set, this match is not valid, so PCRE searches further into the string for occurrences of "a" or "b".

Perl has no direct equivalent of PCRE_NOTEMPTY, but it does make a special case of a pattern match of the empty string within its **split()** function, and when using the /g modifier. It is possible to emulate Perl's behaviour after matching a null string by first trying the match again at the same offset with PCRE_NOTEMPTY and PCRE_ANCHORED, and then if that fails by advancing the starting offset (see below) and trying an ordinary match again. There is some code that demonstrates how to do this in the *pcredemo.c* sample program.

```
  PCRE_NO_UTF8_CHECK
```

When PCRE_UTF8 is set at compile time, the validity of the subject as a UTF-8 string is automatically checked when **pcre_exec()** is subsequently called. The value of *startoffset* is also checked to ensure that it points to the start of a UTF-8 character. If an invalid UTF-8 sequence of bytes is found, **pcre_exec()** returns the error PCRE_ERROR_BADUTF8. If *startoffset* contains an invalid value, PCRE_ERROR_BADUTF8_OFFSET is returned.

If you already know that your subject is valid, and you want to skip these checks for performance reasons, you can set the PCRE_NO_UTF8_CHECK option when calling **pcre_exec()**. You might want to do this for the second and subsequent calls to **pcre_exec()** if you are making repeated calls to find all the matches in a single subject string. However, you should be sure that the value of *startoffset* points to the start of a UTF-8 character. When PCRE_NO_UTF8_CHECK is set, the effect of passing an invalid UTF-8 string as a subject, or a value of *startoffset* that does not point to the start of a UTF-8 character, is undefined. Your program may crash.

```
  PCRE_PARTIAL
```

This option turns on the partial matching feature. If the subject string fails to match the pattern, but at some point during the matching process the end of the subject was reached (that is, the subject partially matches the pattern and the failure to match occurred only because there were not enough subject characters), **pcre_exec()** returns PCRE_ERROR_PARTIAL instead of PCRE_ERROR_NOMATCH. When PCRE_PARTIAL is used, there are restrictions on what may appear in the pattern. These are discussed in the **pcrepartial** documentation.

The string to be matched by pcre_exec()

The subject string is passed to **pcre_exec()** as a pointer in *subject*, a length in *length*, and a starting byte offset in *startoffset*. In UTF-8 mode, the byte offset must point to the start of a UTF-8 character. Unlike the pattern string, the subject may contain binary zero bytes. When the starting offset is zero, the search for a match starts at the beginning of the subject, and this is by far the most common case.

A non-zero starting offset is useful when searching for another match in the same subject by calling **pcre_exec()** again after a previous success. Setting *startoffset* differs from just passing over a shortened string and setting PCRE_NOTBOL in the case of a pattern that begins with any kind of lookbehind. For example, consider the pattern

```
  \Biss\B
```

which finds occurrences of "iss" in the middle of words. (\B matches only if the current position in the subject is not a word boundary.) When applied to the string "Mississipi" the first call to **pcre_exec()** finds the first occurrence. If **pcre_exec()** is called again with just the remainder of the subject, namely "issipi", it does not match, because \B is always false at the start of the subject, which is deemed to be a word boundary. However, if **pcre_exec()** is passed the entire string again, but with *startoffset* set to 4, it finds the second occurrence of "iss" because it is able to look behind the starting point to discover that it is preceded by a letter.

If a non-zero starting offset is passed when the pattern is anchored, one attempt to match at the given offset is made. This can only succeed if the pattern does not require the match to be at the start of the subject.

How pcre_exec() returns captured substrings

In general, a pattern matches a certain portion of the subject, and in addition, further substrings from the subject may be picked out by parts of the pattern. Following the usage in Jeffrey Friedl's book, this is called "capturing" in what follows, and the phrase "capturing subpattern" is used for a fragment of a pattern that picks out a substring. PCRE supports several other kinds of parenthesized subpattern that do not cause substrings to be captured.

Captured substrings are returned to the caller via a vector of integer offsets whose address is passed in *ovector*. The number of elements in the vector is passed in *ovecsize*, which must be a non-negative number. **Note**: this argument is NOT the size of *ovector* in bytes.

The first two-thirds of the vector is used to pass back captured substrings, each substring using a pair of integers. The remaining third of the vector is used as workspace by **pcre_exec()** while matching capturing subpatterns, and is not available for passing back information. The length passed in *ovecsize* should always be a multiple of three. If it is not, it is rounded down.

When a match is successful, information about captured substrings is returned in pairs of integers, starting at the beginning of *ovector*, and continuing up to two-thirds of its length at the most. The first element of a pair is set to the offset of the first character in a substring, and the second is set to the offset of the first character after the end of a substring. The first pair, *ovector[0]* and *ovector[1]*, identify the portion of the subject string matched by the entire pattern. The next pair is used for the first capturing subpattern, and so on. The value returned by **pcre_exec()** is one more than the highest numbered pair that has been set. For example, if two substrings have been captured, the returned value is 3. If there are no capturing subpatterns, the return value from a successful match is 1, indicating that just the first pair of offsets has been set.

If a capturing subpattern is matched repeatedly, it is the last portion of the string that it matched that is returned.

If the vector is too small to hold all the captured substring offsets, it is used as far as possible (up to two-thirds of its length), and the function returns a value of zero. In particular, if the substring offsets are not of interest, **pcre_exec()** may be called with *ovector* passed as NULL and *ovecsize* as zero. However, if the pattern contains back references and the *ovector* is not big enough to remember the related substrings, PCRE has to get additional memory for use during matching. Thus it is usually advisable to supply an *ovector*.

The **pcre_info()** function can be used to find out how many capturing subpatterns there are in a compiled pattern. The smallest size for *ovector* that will allow for *n* captured substrings, in addition to the offsets of the substring matched by the whole pattern, is (*n*+1)*3.

It is possible for capturing subpattern number *n+1* to match some part of the subject when subpattern *n* has not been used at all. For example, if the string "abc" is matched against the pattern (a|(z))(bc) the return from the function is 4, and subpatterns 1 and 3 are matched, but 2 is not. When this happens, both values in the offset pairs corresponding to unused subpatterns are set to -1.

Offset values that correspond to unused subpatterns at the end of the expression are also set to -1. For example, if the string "abc" is matched against the pattern (abc)(x(yz)?)? subpatterns 2 and 3 are not matched. The return from the function is 2, because the highest used capturing subpattern number is 1. However, you can refer to the offsets for the second and third capturing subpatterns if you wish (assuming the vector is large enough, of course).

Some convenience functions are provided for extracting the captured substrings as separate strings. These are described below.

Error return values from pcre_exec()

If **pcre_exec()** fails, it returns a negative number. The following are defined in the header file:

```
  PCRE_ERROR_NOMATCH        (-1)
```

The subject string did not match the pattern.

```
  PCRE_ERROR_NULL           (-2)
```

Either *code* or *subject* was passed as NULL, or *ovector* was NULL and *ovecsize* was not zero.

```
  PCRE_ERROR_BADOPTION      (-3)
```

An unrecognized bit was set in the *options* argument.

```
  PCRE_ERROR_BADMAGIC       (-4)
```

PCRE stores a 4-byte "magic number" at the start of the compiled code, to catch the case when it is passed a junk pointer and to detect when a pattern that was compiled in an environment of one endianness is run in an environment with the other endianness. This is the error that PCRE gives when the magic number is not present.

```
  PCRE_ERROR_UNKNOWN_OPCODE (-5)
```

While running the pattern match, an unknown item was encountered in the compiled pattern. This error could be caused by a bug in PCRE or by overwriting of the compiled pattern.

```
  PCRE_ERROR_NOMEMORY       (-6)
```

If a pattern contains back references, but the *ovector* that is passed to **pcre_exec()** is not big enough to remember the referenced substrings, PCRE gets a block of memory at the start of matching to use for this purpose. If the call via **pcre_malloc()** fails, this error is given. The memory is automatically freed at the end of matching.

```
  PCRE_ERROR_NOSUBSTRING    (-7)
```

This error is used by the **pcre_copy_substring()**, **pcre_get_substring()**, and **pcre_get_substring_list()** functions (see below). It is never returned by **pcre_exec()**.

```
  PCRE_ERROR_MATCHLIMIT     (-8)
```

The backtracking limit, as specified by the *match_limit* field in a **pcre_extra** structure (or defaulted) was reached. See the description above.

```
  PCRE_ERROR_CALLOUT        (-9)
```

This error is never generated by **pcre_exec()** itself. It is provided for use by callout functions that want to yield a distinctive error code. See the **pcrecallout** documentation for details.

```
  PCRE_ERROR_BADUTF8        (-10)
```

A string that contains an invalid UTF-8 byte sequence was passed as a subject.

```
  PCRE_ERROR_BADUTF8_OFFSET (-11)
```

The UTF-8 byte sequence that was passed as a subject was valid, but the value of *startoffset* did not point to the beginning of a UTF-8 character.

```
  PCRE_ERROR_PARTIAL        (-12)
```

The subject string did not match, but it did match partially. See the **pcrepartial** documentation for details of partial matching.

```
  PCRE_ERROR_BADPARTIAL     (-13)
```

The PCRE_PARTIAL option was used with a compiled pattern containing items that are not supported for partial matching. See the **pcrepartial** documentation for details of partial matching.

```
  PCRE_ERROR_INTERNAL       (-14)
```

An unexpected internal error has occurred. This error could be caused by a bug in PCRE or by overwriting of the compiled pattern.

```
  PCRE_ERROR_BADCOUNT       (-15)
```

This error is given if the value of the *ovecsize* argument is negative.

```
  PCRE_ERROR_RECURSIONLIMIT (-21)
```

The internal recursion limit, as specified by the *match_limit_recursion* field in a **pcre_extra** structure (or defaulted) was reached. See the description above.

```
  PCRE_ERROR_NULLWSLIMIT    (-22)
```

When a group that can match an empty substring is repeated with an unbounded upper limit, the subject position at the start of the group must be remembered, so that a test for an empty string can be made when the end of the group is reached. Some workspace is required for this; if it runs out, this error is given.

```
  PCRE_ERROR_BADNEWLINE     (-23)
```

An invalid combination of PCRE_NEWLINE_*xxx* options was given.

Error numbers -16 to -20 are not used by **pcre_exec()**.

EXTRACTING CAPTURED SUBSTRINGS BY NUMBER

int pcre_copy_substring(const char ****subject*, int ****ovector***,**
int *stringcount***, int** *stringnumber***, char ****buffer***,**
int *buffersize***);**

int pcre_get_substring(const char ****subject*, int ****ovector***,**
int *stringcount***, int** *stringnumber***,**
const char *****stringptr*);**

int pcre_get_substring_list(const char ****subject*,**
int ****ovector*, int** *stringcount***, const char ******listptr***);**

Captured substrings can be accessed directly by using the offsets returned by **pcre_exec()** in *ovector*. For convenience, the functions **pcre_copy_substring()**, **pcre_get_substring()**, and **pcre_get_substring_list()** are provided for extracting captured substrings as new, separate, zero-terminated strings. These functions identify substrings by number. The next section describes functions for extracting named substrings.

A substring that contains a binary zero is correctly extracted and has a further zero added on the end, but the result is not, of course, a C string. However, you can process such a string by referring to the length that is returned by **pcre_copy_substring()** and **pcre_get_substring()**. Unfortunately, the interface to **pcre_get_substring_list()** is not adequate for handling strings containing binary zeros, because the end of the final string is not independently indicated.

The first three arguments are the same for all three of these functions: *subject* is the subject string that has just been successfully matched, *ovector* is a pointer to the vector of integer offsets that was passed to **pcre_exec()**, and *stringcount* is the number of substrings that were captured by the match, including the substring that matched the entire regular expression. This is the value returned by **pcre_exec()** if it is greater than zero. If **pcre_exec()** returned zero, indicating that it ran out of space in *ovector*, the value passed as *stringcount* should be the number of elements in the vector divided by three.

The functions **pcre_copy_substring()** and **pcre_get_substring()** extract a single substring, whose number is given as *stringnumber*. A value of zero extracts the substring that matched the entire pattern, whereas higher values extract the captured substrings. For **pcre_copy_substring()**, the string is placed in *buffer*, whose length is given by *buffersize*, while for **pcre_get_substring()** a new block of memory is obtained via **pcre_malloc**, and its address is returned via *stringptr*. The yield of the function is the length of the string, not including the terminating zero, or one of these error codes:

```
  PCRE_ERROR_NOMEMORY       (-6)
```

The buffer was too small for **pcre_copy_substring()**, or the attempt to get memory failed for **pcre_get_substring()**.

```
  PCRE_ERROR_NOSUBSTRING    (-7)
```

There is no substring whose number is *stringnumber*.

The **pcre_get_substring_list()** function extracts all available substrings and builds a list of pointers to them. All this is done in a single block of memory that is obtained via **pcre_malloc**. The address of the memory block is returned via *listptr*, which is also the start of the list of string pointers. The end of the list is marked by a NULL pointer. The yield of the function is zero if all went well, or the error code

```
  PCRE_ERROR_NOMEMORY       (-6)
```

if the attempt to get the memory block failed.

When any of these functions encounter a substring that is unset, which can happen when capturing subpattern number *n+1* matches some part of the subject, but subpattern *n* has not been used at all, they return an empty string. This can be distinguished from a genuine zero-length substring by inspecting the appropriate offset in *ovector*, which is negative for unset substrings.

The two convenience functions **pcre_free_substring()** and **pcre_free_substring_list()** can be used to free the memory returned by a previous call of **pcre_get_substring()** or **pcre_get_substring_list()**, respectively. They do nothing more than call the function pointed to by **pcre_free**, which of course could be called directly from a C program. However, PCRE is used in some situations where it is linked via a special interface to another programming language that cannot use **pcre_free** directly; it is for these cases that the functions are provided.

EXTRACTING CAPTURED SUBSTRINGS BY NAME

int pcre_get_stringnumber(const pcre ****code*,**
        const char ****name*);**

int pcre_copy_named_substring(const pcre ****code*,**
        const char ****subject*, int ****ovector***,**
        int *stringcount***, const char ****stringname***,**
        char ****buffer*, int** *buffersize***);**

int pcre_get_named_substring(const pcre ****code*,**
        const char ****subject*, int ****ovector***,**
        int *stringcount***, const char ****stringname***,**

const char *****stringptr*);**

To extract a substring by name, you first have to find associated number. For example, for this pattern

```
(a+)b(?<xxx>\d+)...
```

the number of the subpattern called "xxx" is 2. If the name is known to be unique (PCRE_DUPNAMES was not set), you can find the number from the name by calling **pcre_get_stringnumber()**. The first argument is the compiled pattern, and the second is the name. The yield of the function is the subpattern number, or PCRE_ERROR_NOSUBSTRING (-7) if there is no subpattern of that name.

Given the number, you can extract the substring directly, or use one of the functions described in the previous section. For convenience, there are also two functions that do the whole job.

Most of the arguments of **pcre_copy_named_substring()** and **pcre_get_named_substring()** are the same as those for the similarly named functions that extract by number. As these are described in the previous section, they are not re-described here. There are just two differences:

First, instead of a substring number, a substring name is given. Second, there is an extra argument, given at the start, which is a pointer to the compiled pattern. This is needed in order to gain access to the name-to-number translation table.

These functions call **pcre_get_stringnumber()**, and if it succeeds, they then call *pcre_copy_substring()* or *pcre_get_substring()*, as appropriate.

DUPLICATE SUBPATTERN NAMES

int pcre_get_stringtable_entries(const pcre ****code*,**
const char ****name*, char *****first***, char *****last***);**

When a pattern is compiled with the PCRE_DUPNAMES option, names for subpatterns are not required to be unique. Normally, patterns with duplicate names are such that in any one match, only one of the named subpatterns participates. An example is shown in the **pcrepattern** documentation. When duplicates are present, **pcre_copy_named_substring()** and **pcre_get_named_substring()** return the first substring corresponding to the given name that is set. If none are set, an empty string is returned. The **pcre_get_stringnumber()** function returns one of the numbers that are associated with the name, but it is not defined which it is.

If you want to get full details of all captured substrings for a given name, you must use the **pcre_get_stringtable_entries()** function. The first argument is the compiled pattern, and the second is the name. The third and fourth are pointers to variables which are updated by the function. After it has run, they point to the first and last entries in the name-to-number table for the given name. The function itself returns the length of each entry, or PCRE_ERROR_NOSUBSTRING (-7) if there are none. The format of the table is described above in the section entitled *Information about a pattern*. Given all the relevant entries for the name, you can extract each of their numbers, and hence the captured data, if any.

FINDING ALL POSSIBLE MATCHES

The traditional matching function uses a similar algorithm to Perl, which stops when it finds the first match, starting at a given point in the subject. If you want to find all possible matches, or the longest possible match, consider using the alternative matching function (see below) instead. If you cannot use the alternative function, but still need to find all possible matches, you can kludge it up by making use of the callout facility, which is described in the **pcrecallout** documentation.

What you have to do is to insert a callout right at the end of the pattern. When your callout function is called, extract and save the current matched substring. Then return 1, which forces **pcre_exec()** to backtrack and try other alternatives. Ultimately, when it runs out of matches, **pcre_exec()** will yield PCRE_ERROR_NOMATCH.

MATCHING A PATTERN: THE ALTERNATIVE FUNCTION

int pcre_dfa_exec(const pcre ****code*, const pcre_extra ****extra***,**
const char ****subject*, int** *length***, int** *startoffset***,**
int *options***, int ****ovector***, int** *ovecsize***,**
int ****workspace*, int** *wscount***);**

The function **pcre_dfa_exec()** is called to match a subject string against a compiled pattern, using a matching algorithm that scans the subject string just once, and does not backtrack. This has different characteristics to the normal algorithm, and is not compatible with Perl. Some of the features of PCRE patterns are not supported. Nevertheless, there are times when this kind of matching can be useful. For a discussion of the two matching algorithms, see the **pcrematching** documentation.

The arguments for the **pcre_dfa_exec()** function are the same as for **pcre_exec()**, plus two extras. The *ovector* argument is used in a different way, and this is described below. The other common arguments are used in the same way as for **pcre_exec()**, so their description is not repeated here.

The two additional arguments provide workspace for the function. The workspace vector should contain at least 20 elements. It is used for keeping track of multiple paths through the pattern tree. More workspace will be needed for patterns and subjects where there are a lot of potential matches.

Here is an example of a simple call to **pcre_dfa_exec()**:

```
int rc;
int ovector[10];
int wspace[20];
rc = pcre_dfa_exec(
  re,             /* result of pcre_compile() */
  NULL,           /* we didn't study the pattern */
  "some string",  /* the subject string */
  11,             /* the length of the subject string */
  0,              /* start at offset 0 in the subject */
  0,              /* default options */
  ovector,        /* vector of integers for substring information */
  10,             /* number of elements (NOT size in bytes) */
  wspace,         /* working space vector */
  20);            /* number of elements (NOT size in bytes) */
```

Option bits for pcre_dfa_exec()

The unused bits of the *options* argument for **pcre_dfa_exec()** must be zero. The only bits that may be set are PCRE_ANCHORED, PCRE_NEWLINE_*xxx*, PCRE_NOTBOL, PCRE_NOTEOL, PCRE_NOTEMPTY, PCRE_NO_UTF8_CHECK, PCRE_PARTIAL, PCRE_DFA_SHORTEST, and PCRE_DFA_RESTART. All but the last three of these are the same as for **pcre_exec()**, so their description is not repeated here.

```
PCRE_PARTIAL
```

This has the same general effect as it does for **pcre_exec()**, but the details are slightly different. When PCRE_PARTIAL is set for **pcre_dfa_exec()**, the return code PCRE_ERROR_NOMATCH is converted into PCRE_ERROR_PARTIAL if the end of the subject is reached, there have been no complete matches, but there is still at least one matching possibility. The portion of the string that provided the partial match is set as the first matching string.

```
PCRE_DFA_SHORTEST
```

Setting the PCRE_DFA_SHORTEST option causes the matching algorithm to stop as soon as it has found one match. Because of the way the alternative algorithm works, this is necessarily the shortest possible match at the first possible matching point in the subject string.

```
PCRE_DFA_RESTART
```

When **pcre_dfa_exec()** is called with the PCRE_PARTIAL option, and returns a partial match, it is possible to call it again, with additional subject characters, and have it continue with the same match. The PCRE_DFA_RESTART option requests this action; when it is set, the *workspace* and *wscount* options must reference the same vector as before because data about the match so far is left in them after a partial match. There is more discussion of this facility in the **pcrepartial** documentation.

Successful returns from pcre_dfa_exec()

When **pcre_dfa_exec()** succeeds, it may have matched more than one substring in the subject. Note, however, that all the matches from one run of the function start at the same point in the subject. The shorter matches are all initial substrings of the longer matches. For example, if the pattern

```
<.*>
```

is matched against the string

```
This is <something> <something else> <something further> no more
```

the three matched strings are

```
<something>
<something> <something else>
<something> <something else> <something further>
```

On success, the yield of the function is a number greater than zero, which is the number of matched substrings. The substrings themselves are returned in *ovector*. Each string uses two elements; the first is the offset to the start, and the second is the offset to the end. In fact, all the strings have the same start offset. (Space could have been saved by giving this only once, but it was decided to retain some compatibility with the way **pcre_exec()** returns data, even though the meaning of the strings is different.)

The strings are returned in reverse order of length; that is, the longest matching string is given first. If there were too many matches to fit into *ovector*, the yield of the function is zero, and the vector is filled with the longest matches.

Error returns from pcre_dfa_exec()

The **pcre_dfa_exec()** function returns a negative number when it fails. Many of the errors are the same as for **pcre_exec()**, and these are described above. There are in addition the following errors that are specific to **pcre_dfa_exec()**:

```
PCRE_ERROR_DFA_UITEM      (-16)
```

This return is given if **pcre_dfa_exec()** encounters an item in the pattern that it does not support, for instance, the use of \C or a back reference.

```
PCRE_ERROR_DFA_UCOND      (-17)
```

This return is given if **pcre_dfa_exec()** encounters a condition item that uses a back reference for the condition, or a test for recursion in a specific group. These are not supported.

```
PCRE_ERROR_DFA_UMLIMIT    (-18)
```

This return is given if **pcre_dfa_exec()** is called with an *extra* block that contains a setting of the *match_limit* field. This is not supported (it is meaningless).

```
PCRE_ERROR_DFA_WSSIZE     (-19)
```

This return is given if **pcre_dfa_exec()** runs out of space in the *workspace* vector.

```
PCRE_ERROR_DFA_RECURSE    (-20)
```

When a recursive subpattern is processed, the matching function calls itself recursively, using private vectors for *ovector* and *workspace*. This error is given if the output vector is not large enough. This should be extremely rare, as a vector of size 1000 is used.

SEE ALSO

pcrebuild(3), **pcrecallout**(3), **pcrecpp(3)**(3), **pcrematching**(3), **pcrepartial**(3), **pcreposix**(3), **pcreprecompile**(3), **pcresample**(3), **pcrestack**(3).

Last updated: 30 November 2006

NAME

PCRE - Perl-compatible regular expressions

PCRE BUILD-TIME OPTIONS

This document describes the optional features of PCRE that can be selected when the library is compiled. They are all selected, or deselected, by providing options to the **configure** script that is run before the **make** command. The complete list of options for **configure** (which includes the standard ones such as the selection of the installation directory) can be obtained by running

```
  ./configure --help
```

The following sections describe certain options whose names begin with --enable or --disable. These settings specify changes to the defaults for the **configure** command. Because of the way that **configure** works, --enable and --disable always come in pairs, so the complementary option always exists as well, but as it specifies the default, it is not described.

C++ SUPPORT

By default, the **configure** script will search for a C++ compiler and C++ header files. If it finds them, it automatically builds the C++ wrapper library for PCRE. You can disable this by adding

```
  --disable-cpp
```

to the **configure** command.

UTF-8 SUPPORT

To build PCRE with support for UTF-8 character strings, add

```
  --enable-utf8
```

to the **configure** command. Of itself, this does not make PCRE treat strings as UTF-8. As well as compiling PCRE with this option, you also have have to set the PCRE_UTF8 option when you call the **pcre_compile()** function.

UNICODE CHARACTER PROPERTY SUPPORT

UTF-8 support allows PCRE to process character values greater than 255 in the strings that it handles. On its own, however, it does not provide any facilities for accessing the properties of such characters. If you want to be able to use the pattern escapes \P, \p, and \X, which refer to Unicode character properties, you must add

```
  --enable-unicode-properties
```

to the **configure** command. This implies UTF-8 support, even if you have not explicitly requested it.

Including Unicode property support adds around 90K of tables to the PCRE library, approximately doubling its size. Only the general category properties such as *Lu* and *Nd* are supported. Details are given in the **pcrepattern** documentation.

CODE VALUE OF NEWLINE

By default, PCRE interprets character 10 (linefeed, LF) as indicating the end of a line. This is the normal newline character on Unix-like systems. You can compile PCRE to use character 13 (carriage return, CR) instead, by adding

```
  --enable-newline-is-cr
```

to the **configure** command. There is also a --enable-newline-is-lf option, which explicitly specifies linefeed as the newline character.

Alternatively, you can specify that line endings are to be indicated by the two character sequence CRLF. If you want this, add

```
  --enable-newline-is-crlf
```

to the **configure** command. There is a fourth option, specified by

```
  --enable-newline-is-any
```

which causes PCRE to recognize any Unicode newline sequence.

Whatever line ending convention is selected when PCRE is built can be overridden when the library functions are called. At build time it is conventional to use the standard for your operating system.

BUILDING SHARED AND STATIC LIBRARIES

The PCRE building process uses **libtool** to build both shared and static Unix libraries by default. You can suppress one of these by adding one of

```
  --disable-shared
  --disable-static
```

to the **configure** command, as required.

POSIX MALLOC USAGE

When PCRE is called through the POSIX interface (see the **pcreposix** documentation), additional working storage is required for holding the pointers to capturing substrings, because PCRE requires three integers per substring, whereas the POSIX interface provides only two. If the number of expected substrings is small, the wrapper function uses space on the stack, because this is faster than using **malloc()** for each call. The default threshold above which the stack is no longer used is 10; it can be changed by adding a setting such as

```
  --with-posix-malloc-threshold=20
```

to the **configure** command.

HANDLING VERY LARGE PATTERNS

Within a compiled pattern, offset values are used to point from one part to another (for example, from an opening parenthesis to an alternation metacharacter). By default, two-byte values are used for these offsets, leading to a maximum size for a compiled pattern of around 64K. This is sufficient to handle all but the most gigantic patterns. Nevertheless, some people do want to process enormous patterns, so it is possible to compile PCRE to use three-byte or four-byte offsets by adding a setting such as

```
  --with-link-size=3
```

to the **configure** command. The value given must be 2, 3, or 4. Using longer offsets slows down the operation of PCRE because it has to load additional bytes when handling them.

If you build PCRE with an increased link size, test 2 (and test 5 if you are using UTF-8) will fail. Part of the output of these tests is a representation of the compiled pattern, and this changes with the link size.

AVOIDING EXCESSIVE STACK USAGE

When matching with the **pcre_exec()** function, PCRE implements backtracking by making recursive calls to an internal function called **match()**. In environments where the size of the stack is limited, this can severely limit PCRE's operation. (The Unix environment does not usually suffer from this problem, but it may sometimes be necessary to increase the maximum stack size. There is a discussion in the **pcrestack** documentation.) An alternative approach to recursion that uses memory from the heap to remember data, instead of using recursive function calls, has been implemented to work round the problem of limited stack size. If you want to build a version of PCRE that works this way, add

```
  --disable-stack-for-recursion
```

to the **configure** command. With this configuration, PCRE will use the **pcre_stack_malloc** and **pcre_stack_free** variables to call memory management functions. Separate functions are provided because the usage is very predictable: the block sizes requested are always the same, and the blocks are always freed in reverse order. A calling program might be able to implement optimized functions that perform better than the standard **malloc()** and **free()** functions. PCRE runs noticeably more slowly when built in this way. This option affects only the **pcre_exec()** function; it is not relevant for the the **pcre_dfa_exec()** function.

LIMITING PCRE RESOURCE USAGE

Internally, PCRE has a function called **match()**, which it calls repeatedly (sometimes recursively) when matching a pattern with the **pcre_exec()** function. By controlling the maximum number of times this function may be called during a single matching operation, a limit can be placed on the resources used by a single call to **pcre_exec()**. The limit can be changed at run time, as described in the **pcreapi** documentation. The default is 10 million, but this can be changed by adding a setting such as

```
  --with-match-limit=500000
```

to the **configure** command. This setting has no effect on the **pcre_dfa_exec()** matching function.

In some environments it is desirable to limit the depth of recursive calls of **match()** more strictly than the total number of calls, in order to restrict the maximum amount of stack (or heap, if --disable-stack-for-recursion is specified) that is used. A second limit controls this; it defaults to the value that is set for --with-match-limit, which imposes no additional constraints. However, you can set a lower limit by adding, for example,

```
  --with-match-limit-recursion=10000
```

to the **configure** command. This value can also be overridden at run time.

USING EBCDIC CODE

PCRE assumes by default that it will run in an environment where the character code is ASCII (or Unicode, which is a superset of ASCII). PCRE can, however, be compiled to run in an EBCDIC environment by adding

```
  --enable-ebcdic
```

to the **configure** command.

SEE ALSO

pcreapi(3), **pcre_config**(3).

Last updated: 30 November 2006

NAME

PCRE - Perl-compatible regular expressions

PCRE CALLOUTS

int (*pcre_callout)(pcre_callout_block *);

PCRE provides a feature called "callout", which is a means of temporarily passing control to the caller of PCRE in the middle of pattern matching. The caller of PCRE provides an external function by putting its entry point in the global variable *pcre_callout*. By default, this variable contains NULL, which disables all calling out.

Within a regular expression, (?C) indicates the points at which the external function is to be called. Different callout points can be identified by putting a number less than 256 after the letter C. The default value is zero. For example, this pattern has two callout points:

```
(?C1)eabc(?C2)def
```

If the PCRE_AUTO_CALLOUT option bit is set when **pcre_compile()** is called, PCRE automatically inserts callouts, all with number 255, before each item in the pattern. For example, if PCRE_AUTO_CALLOUT is used with the pattern

```
A(\d{2}|--)
```

it is processed as if it were

```
(?C255)A(?C255)((?C255)\d{2}(?C255)|(?C255)-(?C255)-(?C255))(?C255)
```

Notice that there is a callout before and after each parenthesis and alternation bar. Automatic callouts can be used for tracking the progress of pattern matching. The **pcretest** command has an option that sets automatic callouts; when it is used, the output indicates how the pattern is matched. This is useful information when you are trying to optimize the performance of a particular pattern.

MISSING CALLOUTS

You should be aware that, because of optimizations in the way PCRE matches patterns, callouts sometimes do not happen. For example, if the pattern is

```
ab(?C4)cd
```

PCRE knows that any matching string must contain the letter "d". If the subject string is "abyz", the lack of "d" means that matching doesn’t ever start, and the callout is never reached. However, with "abyd", though the result is still no match, the callout is obeyed.

THE CALLOUT INTERFACE

During matching, when PCRE reaches a callout point, the external function defined by *pcre_callout* is called (if it is set). This applies to both the **pcre_exec()** and the **pcre_dfa_exec()** matching functions. The only argument to the callout function is a pointer to a **pcre_callout** block. This structure contains the following fields:

```
int          version;
int          callout_number;
int         *offset_vector;
const char  *subject;
int          subject_length;
int          start_match;
int          current_position;
int          capture_top;
int          capture_last;
```

```
  void        *callout_data;
  int         pattern_position;
  int         next_item_length;
```

The *version* field is an integer containing the version number of the block format. The initial version was 0; the current version is 1. The version number will change again in future if additional fields are added, but the intention is never to remove any of the existing fields.

The *callout_number* field contains the number of the callout, as compiled into the pattern (that is, the number after ?C for manual callouts, and 255 for automatically generated callouts).

The *offset_vector* field is a pointer to the vector of offsets that was passed by the caller to **pcre_exec()** or **pcre_dfa_exec()**. When **pcre_exec()** is used, the contents can be inspected in order to extract substrings that have been matched so far, in the same way as for extracting substrings after a match has completed. For **pcre_dfa_exec()** this field is not useful.

The *subject* and *subject_length* fields contain copies of the values that were passed to **pcre_exec()**.

The *start_match* field contains the offset within the subject at which the current match attempt started. If the pattern is not anchored, the callout function may be called several times from the same point in the pattern for different starting points in the subject.

The *current_position* field contains the offset within the subject of the current match pointer.

When the **pcre_exec()** function is used, the *capture_top* field contains one more than the number of the highest numbered captured substring so far. If no substrings have been captured, the value of *capture_top* is one. This is always the case when **pcre_dfa_exec()** is used, because it does not support captured substrings.

The *capture_last* field contains the number of the most recently captured substring. If no substrings have been captured, its value is -1. This is always the case when **pcre_dfa_exec()** is used.

The *callout_data* field contains a value that is passed to **pcre_exec()** or **pcre_dfa_exec()** specifically so that it can be passed back in callouts. It is passed in the *pcre_callout* field of the **pcre_extra** data structure. If no such data was passed, the value of *callout_data* in a **pcre_callout** block is NULL. There is a description of the **pcre_extra** structure in the **pcreapi** documentation.

The *pattern_position* field is present from version 1 of the *pcre_callout* structure. It contains the offset to the next item to be matched in the pattern string.

The *next_item_length* field is present from version 1 of the *pcre_callout* structure. It contains the length of the next item to be matched in the pattern string. When the callout immediately precedes an alternation bar, a closing parenthesis, or the end of the pattern, the length is zero. When the callout precedes an opening parenthesis, the length is that of the entire subpattern.

The *pattern_position* and *next_item_length* fields are intended to help in distinguishing between different automatic callouts, which all have the same callout number. However, they are set for all callouts.

RETURN VALUES

The external callout function returns an integer to PCRE. If the value is zero, matching proceeds as normal. If the value is greater than zero, matching fails at the current point, but the testing of other matching possibilities goes ahead, just as if a lookahead assertion had failed. If the value is less than zero, the match is abandoned, and **pcre_exec()** (or **pcre_dfa_exec()**) returns the negative value.

Negative values should normally be chosen from the set of PCRE_ERROR_xxx values. In particular, PCRE_ERROR_NOMATCH forces a standard "no match" failure. The error number PCRE_ERROR_CALLOUT is reserved for use by callout functions; it will never be used by PCRE itself.

Last updated: 28 February 2005

NAME

PCRE - Perl-compatible regular expressions

DIFFERENCES BETWEEN PCRE AND PERL

This document describes the differences in the ways that PCRE and Perl handle regular expressions. The differences described here are mainly with respect to Perl 5.8, though PCRE version 7.0 contains some features that are expected to be in the forthcoming Perl 5.10.

1. PCRE has only a subset of Perl's UTF-8 and Unicode support. Details of what it does have are given in the section on UTF-8 support in the main **pcre** page.

2. PCRE does not allow repeat quantifiers on lookahead assertions. Perl permits them, but they do not mean what you might think. For example, (?!a){3} does not assert that the next three characters are not "a". It just asserts that the next character is not "a" three times.

3. Capturing subpatterns that occur inside negative lookahead assertions are counted, but their entries in the offsets vector are never set. Perl sets its numerical variables from any such patterns that are matched before the assertion fails to match something (thereby succeeding), but only if the negative lookahead assertion contains just one branch.

4. Though binary zero characters are supported in the subject string, they are not allowed in a pattern string because it is passed as a normal C string, terminated by zero. The escape sequence \0 can be used in the pattern to represent a binary zero.

5. The following Perl escape sequences are not supported: \l, \u, \L, \U, and \N. In fact these are implemented by Perl's general string-handling and are not part of its pattern matching engine. If any of these are encountered by PCRE, an error is generated.

6. The Perl escape sequences \p, \P, and \X are supported only if PCRE is built with Unicode character property support. The properties that can be tested with \p and \P are limited to the general category properties such as Lu and Nd, script names such as Greek or Han, and the derived properties Any and L&.

7. PCRE does support the \Q...\E escape for quoting substrings. Characters in between are treated as literals. This is slightly different from Perl in that $ and @ are also handled as literals inside the quotes. In Perl, they cause variable interpolation (but of course PCRE does not have variables). Note the following examples:

```
  Pattern            PCRE matches      Perl matches

  \Qabc$xyz\E        abc$xyz           abc followed by the
                                          contents of $xyz
  \Qabc\$xyz\E       abc\$xyz          abc\$xyz
  \Qabc\E\$\Qxyz\E   abc$xyz           abc$xyz
```

The \Q...\E sequence is recognized both inside and outside character classes.

8. Fairly obviously, PCRE does not support the (?{code}) and (??{code}) constructions. However, there is support for recursive patterns. This is not available in Perl 5.8, but will be in Perl 5.10. Also, the PCRE "callout" feature allows an external function to be called during pattern matching. See the **pcrecallout** documentation for details.

9. Subpatterns that are called recursively or as "subroutines" are always treated as atomic groups in PCRE. This is like Python, but unlike Perl.

10. There are some differences that are concerned with the settings of captured strings when part of a pattern is repeated. For example, matching "aba" against the pattern /^(a(b)?)+$/ in Perl leaves $2 unset, but in PCRE it is set to "b".

11. PCRE provides some extensions to the Perl regular expression facilities. Perl 5.10 will include new features that are not in earlier versions, some of which (such as named parentheses) have been in PCRE for some time. This list is with respect to Perl 5.10:

(a) Although lookbehind assertions must match fixed length strings, each alternative branch of a lookbehind assertion can match a different length of string. Perl requires them all to have the same length.

(b) If PCRE_DOLLAR_ENDONLY is set and PCRE_MULTILINE is not set, the $ meta-character matches only at the very end of the string.

(c) If PCRE_EXTRA is set, a backslash followed by a letter with no special meaning is faulted. Otherwise, like Perl, the backslash is ignored. (Perl can be made to issue a warning.)

(d) If PCRE_UNGREEDY is set, the greediness of the repetition quantifiers is inverted, that is, by default they are not greedy, but if followed by a question mark they are.

(e) PCRE_ANCHORED can be used at matching time to force a pattern to be tried only at the first matching position in the subject string.

(f) The PCRE_NOTBOL, PCRE_NOTEOL, PCRE_NOTEMPTY, and PCRE_NO_AUTO_CAPTURE options for **pcre_exec()** have no Perl equivalents.

(g) The callout facility is PCRE-specific.

(h) The partial matching facility is PCRE-specific.

(i) Patterns compiled by PCRE can be saved and re-used at a later time, even on different hosts that have the other endianness.

(j) The alternative matching function (**pcre_dfa_exec()**) matches in a different way and is not Perl-compatible.

Last updated: 28 November 2006

NAME

PCRE - Perl-compatible regular expressions

PCRE MATCHING ALGORITHMS

This document describes the two different algorithms that are available in PCRE for matching a compiled regular expression against a given subject string. The "standard" algorithm is the one provided by the **pcre_exec()** function. This works in the same was as Perl's matching function, and provides a Perl-compatible matching operation.

An alternative algorithm is provided by the **pcre_dfa_exec()** function; this operates in a different way, and is not Perl-compatible. It has advantages and disadvantages compared with the standard algorithm, and these are described below.

When there is only one possible way in which a given subject string can match a pattern, the two algorithms give the same answer. A difference arises, however, when there are multiple possibilities. For example, if the pattern

```
  ^<.*>
```

is matched against the string

```
  <something> <something else> <something further>
```

there are three possible answers. The standard algorithm finds only one of them, whereas the alternative algorithm finds all three.

REGULAR EXPRESSIONS AS TREES

The set of strings that are matched by a regular expression can be represented as a tree structure. An unlimited repetition in the pattern makes the tree of infinite size, but it is still a tree. Matching the pattern to a given subject string (from a given starting point) can be thought of as a search of the tree. There are two ways to search a tree: depth-first and breadth-first, and these correspond to the two matching algorithms provided by PCRE.

THE STANDARD MATCHING ALGORITHM

In the terminology of Jeffrey Friedl's book *Mastering Regular Expressions*, the standard algorithm is an "NFA algorithm". It conducts a depth-first search of the pattern tree. That is, it proceeds along a single path through the tree, checking that the subject matches what is required. When there is a mismatch, the algorithm tries any alternatives at the current point, and if they all fail, it backs up to the previous branch point in the tree, and tries the next alternative branch at that level. This often involves backing up (moving to the left) in the subject string as well. The order in which repetition branches are tried is controlled by the greedy or ungreedy nature of the quantifier.

If a leaf node is reached, a matching string has been found, and at that point the algorithm stops. Thus, if there is more than one possible match, this algorithm returns the first one that it finds. Whether this is the shortest, the longest, or some intermediate length depends on the way the greedy and ungreedy repetition quantifiers are specified in the pattern.

Because it ends up with a single path through the tree, it is relatively straightforward for this algorithm to keep track of the substrings that are matched by portions of the pattern in parentheses. This provides support for capturing parentheses and back references.

THE ALTERNATIVE MATCHING ALGORITHM

This algorithm conducts a breadth-first search of the tree. Starting from the first matching point in the subject, it scans the subject string from left to right, once, character by character, and as it does this, it remembers all the paths through the tree that represent valid matches. In Friedl's terminology, this is a kind of "DFA algorithm", though it is not implemented as a traditional finite state machine (it keeps multiple states active simultaneously).

The scan continues until either the end of the subject is reached, or there are no more unterminated paths. At this point, terminated paths represent the different matching possibilities (if there are none, the match has failed). Thus, if there is more than one possible match, this algorithm finds all of them, and in particular, it finds the longest. In PCRE, there is an option to stop the algorithm after the first match (which is necessarily the shortest) has been found.

Note that all the matches that are found start at the same point in the subject. If the pattern

```
  cat(er(pillar)?)
```

is matched against the string "the caterpillar catchment", the result will be the three strings "cat", "cater", and "caterpillar" that start at the fourth character of the subject. The algorithm does not automatically move on to find matches that start at later positions.

There are a number of features of PCRE regular expressions that are not supported by the alternative matching algorithm. They are as follows:

1. Because the algorithm finds all possible matches, the greedy or ungreedy nature of repetition quantifiers is not relevant. Greedy and ungreedy quantifiers are treated in exactly the same way. However, possessive quantifiers can make a difference when what follows could also match what is quantified, for example in a pattern like this:

```
  ˆa++\w!
```

This pattern matches "aaab!" but not "aaa!", which would be matched by a non-possessive quantifier. Similarly, if an atomic group is present, it is matched as if it were a standalone pattern at the current point, and the longest match is then "locked in" for the rest of the overall pattern.

2. When dealing with multiple paths through the tree simultaneously, it is not straightforward to keep track of captured substrings for the different matching possibilities, and PCRE's implementation of this algorithm does not attempt to do this. This means that no captured substrings are available.

3. Because no substrings are captured, back references within the pattern are not supported, and cause errors if encountered.

4. For the same reason, conditional expressions that use a backreference as the condition or test for a specific group recursion are not supported.

5. Callouts are supported, but the value of the *capture_top* field is always 1, and the value of the *capture_last* field is always -1.

6. The \C escape sequence, which (in the standard algorithm) matches a single byte, even in UTF-8 mode, is not supported because the alternative algorithm moves through the subject string one character at a time, for all active paths through the tree.

ADVANTAGES OF THE ALTERNATIVE ALGORITHM

Using the alternative matching algorithm provides the following advantages:

1. All possible matches (at a single point in the subject) are automatically found, and in particular, the longest match is found. To find more than one match using the standard algorithm, you have to do kludgy things with callouts.

2. There is much better support for partial matching. The restrictions on the content of the pattern that apply when using the standard algorithm for partial matching do not apply to the alternative algorithm. For non-anchored patterns, the starting position of a partial match is available.

3. Because the alternative algorithm scans the subject string just once, and never needs to backtrack, it is possible to pass very long subject strings to the matching function in several pieces, checking for partial matching each time.

DISADVANTAGES OF THE ALTERNATIVE ALGORITHM

The alternative algorithm suffers from a number of disadvantages:

1. It is substantially slower than the standard algorithm. This is partly because it has to search for all possible matches, but is also because it is less susceptible to optimization.

2. Capturing parentheses and back references are not supported.

3. Although atomic groups are supported, their use does not provide the performance advantage that it does for the standard algorithm.

Last updated: 24 November 2006

NAME

PCRE - Perl-compatible regular expressions

PARTIAL MATCHING IN PCRE

In normal use of PCRE, if the subject string that is passed to **pcre_exec()** or **pcre_dfa_exec()** matches as far as it goes, but is too short to match the entire pattern, PCRE_ERROR_NOMATCH is returned. There are circumstances where it might be helpful to distinguish this case from other cases in which there is no match.

Consider, for example, an application where a human is required to type in data for a field with specific formatting requirements. An example might be a date in the form *ddmmmyy*, defined by this pattern:

```
^\d?\d(jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)\d\d$
```

If the application sees the user's keystrokes one by one, and can check that what has been typed so far is potentially valid, it is able to raise an error as soon as a mistake is made, possibly beeping and not reflecting the character that has been typed. This immediate feedback is likely to be a better user interface than a check that is delayed until the entire string has been entered.

PCRE supports the concept of partial matching by means of the PCRE_PARTIAL option, which can be set when calling **pcre_exec()** or **pcre_dfa_exec()**. When this flag is set for **pcre_exec()**, the return code PCRE_ERROR_NOMATCH is converted into PCRE_ERROR_PARTIAL if at any time during the matching process the last part of the subject string matched part of the pattern. Unfortunately, for non-anchored matching, it is not possible to obtain the position of the start of the partial match. No captured data is set when PCRE_ERROR_PARTIAL is returned.

When PCRE_PARTIAL is set for **pcre_dfa_exec()**, the return code PCRE_ERROR_NOMATCH is converted into PCRE_ERROR_PARTIAL if the end of the subject is reached, there have been no complete matches, but there is still at least one matching possibility. The portion of the string that provided the partial match is set as the first matching string.

Using PCRE_PARTIAL disables one of PCRE's optimizations. PCRE remembers the last literal byte in a pattern, and abandons matching immediately if such a byte is not present in the subject string. This optimization cannot be used for a subject string that might match only partially.

RESTRICTED PATTERNS FOR PCRE_PARTIAL

Because of the way certain internal optimizations are implemented in the **pcre_exec()** function, the PCRE_PARTIAL option cannot be used with all patterns. These restrictions do not apply when **pcre_dfa_exec()** is used. For **pcre_exec()**, repeated single characters such as

```
a{2,4}
```

and repeated single metasequences such as

```
\d+
```

are not permitted if the maximum number of occurrences is greater than one. Optional items such as \d? (where the maximum is one) are permitted. Quantifiers with any values are permitted after parentheses, so the invalid examples above can be coded thus:

```
(a){2,4}
(\d)+
```

These constructions run more slowly, but for the kinds of application that are envisaged for this facility, this is not felt to be a major restriction.

If PCRE_PARTIAL is set for a pattern that does not conform to the restrictions, **pcre_exec()** returns the error code PCRE_ERROR_BADPARTIAL (-13).

EXAMPLE OF PARTIAL MATCHING USING PCRETEST

If the escape sequence \P is present in a **pcretest** data line, the PCRE_PARTIAL flag is used for the match. Here is a run of **pcretest** that uses the date example quoted above:

```
  re> /^\d?\d(jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)\d\d$/
data> 25jun04\P
 0: 25jun04
 1: jun
data> 25dec3\P
Partial match
data> 3ju\P
Partial match
data> 3juj\P
No match
data> j\P
No match
```

The first data string is matched completely, so **pcretest** shows the matched substrings. The remaining four strings do not match the complete pattern, but the first two are partial matches. The same test, using **pcre_dfa_exec()** matching (by means of the \D escape sequence), produces the following output:

```
  re> /^?(jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)$/
data> 25jun04\P\D
 0: 25jun04
data> 23dec3\P\D
Partial match: 23dec3
data> 3ju\P\D
Partial match: 3ju
data> 3juj\P\D
No match
data> j\P\D
No match
```

Notice that in this case the portion of the string that was matched is made available.

MULTI-SEGMENT MATCHING WITH pcre_dfa_exec()

When a partial match has been found using **pcre_dfa_exec()**, it is possible to continue the match by providing additional subject data and calling **pcre_dfa_exec()** again with the same compiled regular expression, this time setting the PCRE_DFA_RESTART option. You must also pass the same working space as before, because this is where details of the previous partial match are stored. Here is an example using **pcretest**, using the \R escape sequence to set the PCRE_DFA_RESTART option (\P and \D are as above):

```
  re> /^?(jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)$/
data> 23ja\P\D
Partial match: 23ja
data> n05\R\D
 0: n05
```

The first call has "23ja" as the subject, and requests partial matching; the second call has "n05" as the subject for the continued (restarted) match. Notice that when the match is complete, only the last part is shown; PCRE does not retain the previously partially-matched string. It is up to the calling program to do that if it needs to.

You can set PCRE_PARTIAL with PCRE_DFA_RESTART to continue partial matching over multiple segments. This facility can be used to pass very long subject strings to **pcre_dfa_exec()**. However, some care is needed for certain types of pattern.

1. If the pattern contains tests for the beginning or end of a line, you need to pass the PCRE_NOTBOL or PCRE_NOTEOL options, as appropriate, when the subject string for any call does not contain the beginning or end of a line.

2. If the pattern contains backward assertions (including \b or \B), you need to arrange for some overlap in the subject strings to allow for this. For example, you could pass the subject in chunks that are 500 bytes long, but in a buffer of 700 bytes, with the starting offset set to 200 and the previous 200 bytes at the start of the buffer.

3. Matching a subject string that is split into multiple segments does not always produce exactly the same result as matching over one single long string. The difference arises when there are multiple matching possibilities, because a partial match result is given only when there are no completed matches in a call to fBpcre_dfa_exec(). This means that as soon as the shortest match has been found, continuation to a new subject segment is no longer possible. Consider this **pcretest** example:

```
    re> /dog(sbody)?/
  data> do\P\D
  Partial match: do
  data> gsb\R\P\D
   0: g
  data> dogsbody\D
   0: dogsbody
   1: dog
```

The pattern matches the words "dog" or "dogsbody". When the subject is presented in several parts ("do" and "gsb" being the first two) the match stops when "dog" has been found, and it is not possible to continue. On the other hand, if "dogsbody" is presented as a single string, both matches are found.

Because of this phenomenon, it does not usually make sense to end a pattern that is going to be matched in this way with a variable repeat.

4. Patterns that contain alternatives at the top level which do not all start with the same pattern item may not work as expected. For example, consider this pattern:

```
  1234|3789
```

If the first part of the subject is "ABC123", a partial match of the first alternative is found at offset 3. There is no partial match for the second alternative, because such a match does not start at the same point in the subject string. Attempting to continue with the string "789" does not yield a match because only those alternatives that match at one point in the subject are remembered. The problem arises because the start of the second alternative matches within the first alternative. There is no problem with anchored patterns or patterns such as:

```
  1234|ABCD
```

where no string can be a partial match for both alternatives.

Last updated: 30 November 2006

NAME

PCRE - Perl-compatible regular expressions

PCRE REGULAR EXPRESSION DETAILS

The syntax and semantics of the regular expressions supported by PCRE are described below. Regular expressions are also described in the Perl documentation and in a number of books, some of which have copious examples. Jeffrey Friedl’s "Mastering Regular Expressions", published by O’Reilly, covers regular expressions in great detail. This description of PCRE’s regular expressions is intended as reference material.

The original operation of PCRE was on strings of one-byte characters. However, there is now also support for UTF-8 character strings. To use this, you must build PCRE to include UTF-8 support, and then call **pcre_compile()** with the PCRE_UTF8 option. How this affects pattern matching is mentioned in several places below. There is also a summary of UTF-8 features in the section on UTF-8 support in the main **pcre** page.

The remainder of this document discusses the patterns that are supported by PCRE when its main matching function, **pcre_exec()**, is used. From release 6.0, PCRE offers a second matching function, **pcre_dfa_exec()**, which matches using a different algorithm that is not Perl-compatible. The advantages and disadvantages of the alternative function, and how it differs from the normal function, are discussed in the **pcrematching** page.

CHARACTERS AND METACHARACTERS

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern

```
 The quick brown fox
```

matches a portion of a subject string that is identical to itself. When caseless matching is specified (the PCRE_CASELESS option), letters are matched independently of case. In UTF-8 mode, PCRE always understands the concept of case for characters whose values are less than 128, so caseless matching is always possible. For characters with higher values, the concept of case is supported if PCRE is compiled with Unicode property support, but not otherwise. If you want to use caseless matching for characters 128 and above, you must ensure that PCRE is compiled with Unicode property support as well as with UTF-8 support.

The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of *metacharacters*, which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of metacharacters: those that are recognized anywhere in the pattern except within square brackets, and those that are recognized within square brackets. Outside square brackets, the metacharacters are as follows:

```
 \    general escape character with several uses
 ^    assert start of string (or line, in multiline mode)
 $    assert end of string (or line, in multiline mode)
 .    match any character except newline (by default)
 [    start character class definition
 |    start of alternative branch
 (    start subpattern
 )    end subpattern
 ?    extends the meaning of (
      also 0 or 1 quantifier
      also quantifier minimizer
 *    0 or more quantifier
 +    1 or more quantifier
      also "possessive quantifier"
```

```
  {      start min/max quantifier
```

Part of a pattern that is in square brackets is called a "character class". In a character class the only metacharacters are:

```
  \      general escape character
  ^      negate the class, but only if the first character
  -      indicates character range
  [      POSIX character class (only if followed by POSIX
           syntax)
  ]      terminates the character class
```

The following sections describe the use of each of the metacharacters.

BACKSLASH

The backslash character has several uses. Firstly, if it is followed by a non-alphanumeric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, if you want to match a * character, you write * in the pattern. This escaping action applies whether or not the following character would otherwise be interpreted as a metacharacter, so it is always safe to precede a non-alphanumeric with backslash to specify that it stands for itself. In particular, if you want to match a backslash, you write \\.

If a pattern is compiled with the PCRE_EXTENDED option, whitespace in the pattern (other than in a character class) and characters between a # outside a character class and the next newline are ignored. An escaping backslash can be used to include a whitespace or # character as part of the pattern.

If you want to remove the special meaning from a sequence of characters, you can do so by putting them between \Q and \E. This is different from Perl in that $ and @ are handled as literals in \Q...\E sequences in PCRE, whereas in Perl, $ and @ cause variable interpolation. Note the following examples:

```
  Pattern            PCRE matches   Perl matches

  \Qabc$xyz\E        abc$xyz        abc followed by the
                                      contents of $xyz
  \Qabc\$xyz\E       abc\$xyz       abc\$xyz
  \Qabc\E\$\Qxyz\E   abc$xyz        abc$xyz
```

The \Q...\E sequence is recognized both inside and outside character classes.

Non-printing characters

A second use of backslash provides a way of encoding non-printing characters in patterns in a visible manner. There is no restriction on the appearance of non-printing characters, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is usually easier to use one of the following escape sequences than the binary character it represents:

```
  \a        alarm, that is, the BEL character (hex 07)
  \cx       "control-x", where x is any character
  \e        escape (hex 1B)
  \f        formfeed (hex 0C)
  \n        newline (hex 0A)
  \r        carriage return (hex 0D)
  \t        tab (hex 09)
  \ddd      character with octal code ddd, or backreference
  \xhh      character with hex code hh
  \x{hhh..} character with hex code hhh..
```

The precise effect of \cx is as follows: if x is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus \cz becomes hex 1A, but \c{ becomes hex 3B, while \c; becomes hex 7B.

After \x, from zero to two hexadecimal digits are read (letters can be in upper or lower case). Any number of hexadecimal digits may appear between \x{ and }, but the value of the character code must be less than 256 in non-UTF-8 mode, and less than 2**31 in UTF-8 mode (that is, the maximum hexadecimal value is 7FFFFFFF). If characters other than hexadecimal digits appear between \x{ and }, or if there is no terminating }, this form of escape is not recognized. Instead, the initial \x will be interpreted as a basic hexadecimal escape, with no following digits, giving a character whose value is zero.

Characters whose value is less than 256 can be defined by either of the two syntaxes for \x. There is no difference in the way they are handled. For example, \xdc is exactly the same as \x{dc}.

After \0 up to two further octal digits are read. If there are fewer than two digits, just those that are present are used. Thus the sequence \0\x\07 specifies two binary zeros followed by a BEL character (code value 7). Make sure you supply two digits after the initial zero if the pattern character that follows is itself an octal digit.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, PCRE reads it and any following digits as a decimal number. If the number is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a *back reference*. A description of how this works is given later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, PCRE re-reads up to three octal digits following the backslash, and uses them to generate a data character. Any subsequent digits stand for themselves. In non-UTF-8 mode, the value of a character specified in octal must be less than \400. In UTF-8 mode, values up to \777 are permitted. For example:

```
\040   is another way of writing a space
\40    is the same, provided there are fewer than 40
         previous capturing subpatterns
\7     is always a back reference
\11    might be a back reference, or another way of
         writing a tab
\011   is always a tab
\0113  is a tab followed by the character "3"
\113   might be a back reference, otherwise the
         character with octal code 113
\377   might be a back reference, otherwise
         the byte consisting entirely of 1 bits
\81    is either a back reference, or a binary zero
         followed by the two characters "8" and "1"
```

Note that octal values of 100 or greater must not be introduced by a leading zero, because no more than three octal digits are ever read.

All the sequences that define a single character value can be used both inside and outside character classes. In addition, inside a character class, the sequence \b is interpreted as the backspace character (hex 08), and the sequences \R and \X are interpreted as the characters "R" and "X", respectively. Outside a character class, these sequences have different meanings (see below).

Absolute and relative back references

The sequence \g followed by a positive or negative number, optionally enclosed in braces, is an absolute or relative back reference. Back references are discussed later, following the discussion of parenthesized subpatterns.

Generic character types

Another use of backslash is for specifying generic character types. The following are always recognized:

```
\d    any decimal digit
\D    any character that is not a decimal digit
\s    any whitespace character
\S    any character that is not a whitespace character
\w    any "word" character
\W    any "non-word" character
```

Each pair of escape sequences partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair.

These character type sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all of them fail, since there is no character to match.

For compatibility with Perl, \s does not match the VT character (code 11). This makes it different from the the POSIX "space" class. The \s characters are HT (9), LF (10), FF (12), CR (13), and space (32). (If "use locale;" is included in a Perl script, \s may match the VT character. In PCRE, it never does.)

A "word" character is an underscore or any character less than 256 that is a letter or digit. The definition of letters and digits is controlled by PCRE's low-valued character tables, and may vary if locale-specific matching is taking place (see "Locale support" in the **pcreapi** page). For example, in the "fr_FR" (French) locale, some character codes greater than 128 are used for accented letters, and these are matched by \w.

In UTF-8 mode, characters with values greater than 128 never match \d, \s, or \w, and always match \D, \S, and \W. This is true even when Unicode character property support is available. The use of locales with Unicode is discouraged.

Newline sequences

Outside a character class, the escape sequence \R matches any Unicode newline sequence. This is an extension to Perl. In non-UTF-8 mode \R is equivalent to the following:

```
(?>\r\n|\n|\x0b|\f|\r|\x85)
```

This is an example of an "atomic group", details of which are given below. This particular group matches either the two-character sequence CR followed by LF, or one of the single characters LF (linefeed, U+000A), VT (vertical tab, U+000B), FF (formfeed, U+000C), CR (carriage return, U+000D), or NEL (next line, U+0085). The two-character sequence is treated as a single unit that cannot be split.

In UTF-8 mode, two additional characters whose codepoints are greater than 255 are added: LS (line separator, U+2028) and PS (paragraph separator, U+2029). Unicode character property support is not needed for these characters to be recognized.

Inside a character class, \R matches the letter "R".

Unicode character properties

When PCRE is built with Unicode character property support, three additional escape sequences to match character properties are available when UTF-8 mode is selected. They are:

\p{*xx*}	a character with the *xx* property
\P{*xx*}	a character without the *xx* property
\X	an extended Unicode sequence

The property names represented by *xx* above are limited to the Unicode script names, the general category properties, and "Any", which matches any character (including newline). Other properties such as "InMusicalSymbols" are not currently supported by PCRE. Note that \P{Any} does not match any characters, so always causes a match failure.

Sets of Unicode characters are defined as belonging to certain scripts. A character from one of these sets can be matched using a script name. For example:

```
\p{Greek}
```

```
\P{Han}
```

Those that are not part of an identified script are lumped together as "Common". The current list of scripts is:

Arabic, Armenian, Balinese, Bengali, Bopomofo, Braille, Buginese, Buhid, Canadian_Aboriginal, Cherokee, Common, Coptic, Cuneiform, Cypriot, Cyrillic, Deseret, Devanagari, Ethiopic, Georgian, Glagolitic, Gothic, Greek, Gujarati, Gurmukhi, Han, Hangul, Hanunoo, Hebrew, Hiragana, Inherited, Kannada, Katakana, Kharoshthi, Khmer, Lao, Latin, Limbu, Linear_B, Malayalam, Mongolian, Myanmar, New_Tai_Lue, Nko, Ogham, Old_Italic, Old_Persian, Oriya, Osmanya, Phags_Pa, Phoenician, Runic, Shavian, Sinhala, Syloti_Nagri, Syriac, Tagalog, Tagbanwa, Tai_Le, Tamil, Telugu, Thaana, Thai, Tibetan, Tifinagh, Ugaritic, Yi.

Each character has exactly one general category property, specified by a two-letter abbreviation. For compatibility with Perl, negation can be specified by including a circumflex between the opening brace and the property name. For example, \p{^Lu} is the same as \P{Lu}.

If only one letter is specified with \p or \P, it includes all the general category properties that start with that letter. In this case, in the absence of negation, the curly brackets in the escape sequence are optional; these two examples have the same effect:

```
\p{L}
\pL
```

The following general category property codes are supported:

```
C     Other
Cc    Control
Cf    Format
Cn    Unassigned
Co    Private use
Cs    Surrogate

L     Letter
Ll    Lower case letter
Lm    Modifier letter
Lo    Other letter
Lt    Title case letter
Lu    Upper case letter

M     Mark
Mc    Spacing mark
Me    Enclosing mark
Mn    Non-spacing mark

N     Number
Nd    Decimal number
Nl    Letter number
No    Other number

P     Punctuation
Pc    Connector punctuation
Pd    Dash punctuation
Pe    Close punctuation
Pf    Final punctuation
Pi    Initial punctuation
Po    Other punctuation
Ps    Open punctuation

S     Symbol
```

```
Sc    Currency symbol
Sk    Modifier symbol
Sm    Mathematical symbol
So    Other symbol

Z     Separator
Zl    Line separator
Zp    Paragraph separator
Zs    Space separator
```

The special property L& is also supported: it matches a character that has the Lu, Ll, or Lt property, in other words, a letter that is not classified as a modifier or "other".

The long synonyms for these properties that Perl supports (such as \p{Letter}) are not supported by PCRE, nor is it permitted to prefix any of these properties with "Is".

No character that is in the Unicode table has the Cn (unassigned) property. Instead, this property is assumed for any code point that is not in the Unicode table.

Specifying caseless matching does not affect these escape sequences. For example, \p{Lu} always matches only upper case letters.

The \X escape matches any number of Unicode characters that form an extended Unicode sequence. \X is equivalent to

```
(?>\PM\pM*)
```

That is, it matches a character without the "mark" property, followed by zero or more characters with the "mark" property, and treats the sequence as an atomic group (see below). Characters with the "mark" property are typically accents that affect the preceding character.

Matching characters by Unicode property is not fast, because PCRE has to search a structure that contains data for over fifteen thousand characters. That is why the traditional escape sequences such as \d and \w do not use Unicode properties in PCRE.

Simple assertions

The final use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match, without consuming any characters from the subject string. The use of subpatterns for more complicated assertions is described below. The backslashed assertions are:

```
\b    matches at a word boundary
\B    matches when not at a word boundary
\A    matches at the start of the subject
\Z    matches at the end of the subject
      also matches before a newline at the end of the subject
\z    matches only at the end of the subject
\G    matches at the first matching position in the subject
```

These assertions may not appear in character classes (but note that \b has a different meaning, namely the backspace character, inside a character class).

A word boundary is a position in the subject string where the current character and the previous character do not both match \w or \W (i.e. one matches \w and the other matches \W), or the start or end of the string if the first or last character matches \w, respectively.

The \A, \Z, and \z assertions differ from the traditional circumflex and dollar (described in the next section) in that they only ever match at the very start and end of the subject string, whatever options are set. Thus, they are independent of multiline mode. These three assertions are not affected by the PCRE_NOTBOL or PCRE_NOTEOL options, which affect only the behaviour of the circumflex and dollar metacharacters. However, if the *startoffset* argument of **pcre_exec()** is non-zero, indicating that matching is to start at a point other than the beginning of the subject, \A can never match. The difference between \Z and \z is that \Z matches before a newline at the end of the string as well as at the very end, whereas \z matches only at the end.

The \G assertion is true only when the current matching position is at the start point of the match, as specified by the *startoffset* argument of **pcre_exec()**. It differs from \A when the value of *startoffset* is non-zero. By calling **pcre_exec()** multiple times with appropriate arguments, you can mimic Perl's /g option, and it is in this kind of implementation where \G can be useful.

Note, however, that PCRE's interpretation of \G, as the start of the current match, is subtly different from Perl's, which defines it as the end of the previous match. In Perl, these can be different when the previously matched string was empty. Because PCRE does just one match at a time, it cannot reproduce this behaviour.

If all the alternatives of a pattern begin with \G, the expression is anchored to the starting match position, and the "anchored" flag is set in the compiled regular expression.

CIRCUMFLEX AND DOLLAR

Outside a character class, in the default matching mode, the circumflex character is an assertion that is true only if the current matching point is at the start of the subject string. If the *startoffset* argument of **pcre_exec()** is non-zero, circumflex can never match if the PCRE_MULTILINE option is unset. Inside a character class, circumflex has an entirely different meaning (see below).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative in which it appears if the pattern is ever to match that branch. If all possible alternatives start with a circumflex, that is, if the pattern is constrained to match only at the start of the subject, it is said to be an "anchored" pattern. (There are also other constructs that can cause a pattern to be anchored.)

A dollar character is an assertion that is true only if the current matching point is at the end of the subject string, or immediately before a newline at the end of the string (by default). Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch in which it appears. Dollar has no special meaning in a character class.

The meaning of dollar can be changed so that it matches only at the very end of the string, by setting the PCRE_DOLLAR_ENDONLY option at compile time. This does not affect the \Z assertion.

The meanings of the circumflex and dollar characters are changed if the PCRE_MULTILINE option is set. When this is the case, a circumflex matches immediately after internal newlines as well as at the start of the subject string. It does not match after a newline that ends the string. A dollar matches before any newlines in the string, as well as at the very end, when PCRE_MULTILINE is set. When newline is specified as the two-character sequence CRLF, isolated CR and LF characters do not indicate newlines.

For example, the pattern /^abc$/ matches the subject string "def\nabc" (where \n represents a newline) in multiline mode, but not otherwise. Consequently, patterns that are anchored in single line mode because all branches start with ^ are not anchored in multiline mode, and a match for circumflex is possible when the *startoffset* argument of **pcre_exec()** is non-zero. The PCRE_DOLLAR_ENDONLY option is ignored if PCRE_MULTILINE is set.

Note that the sequences \A, \Z, and \z can be used to match the start and end of the subject in both modes, and if all branches of a pattern start with \A it is always anchored, whether or not PCRE_MULTILINE is set.

FULL STOP (PERIOD, DOT)

Outside a character class, a dot in the pattern matches any one character in the subject string except (by default) a character that signifies the end of a line. In UTF-8 mode, the matched character may be more than one byte long.

When a line ending is defined as a single character, dot never matches that character; when the two-character sequence CRLF is used, dot does not match CR if it is immediately followed by LF, but otherwise it matches all characters (including isolated CRs and LFs). When any Unicode line endings are being recognized, dot does not match CR or LF or any of the other line ending characters.

The behaviour of dot with regard to newlines can be changed. If the PCRE_DOTALL option is set, a dot matches any one character, without exception. If the two-character sequence CRLF is present in the subject string, it takes two dots to match it.

The handling of dot is entirely independent of the handling of circumflex and dollar, the only relationship being that they both involve newlines. Dot has no special meaning in a character class.

MATCHING A SINGLE BYTE

Outside a character class, the escape sequence \C matches any one byte, both in and out of UTF-8 mode. Unlike a dot, it always matches any line-ending characters. The feature is provided in Perl in order to match individual bytes in UTF-8 mode. Because it breaks up UTF-8 characters into individual bytes, what remains in the string may be a malformed UTF-8 string. For this reason, the \C escape sequence is best avoided.

PCRE does not allow \C to appear in lookbehind assertions (described below), because in UTF-8 mode this would make it impossible to calculate the length of the lookbehind.

SQUARE BRACKETS AND CHARACTER CLASSES

An opening square bracket introduces a character class, terminated by a closing square bracket. A closing square bracket on its own is not special. If a closing square bracket is required as a member of the class, it should be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

A character class matches a single character in the subject. In UTF-8 mode, the character may occupy more than one byte. A matched character must be in the set of characters defined by the class, unless the first character in the class definition is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

For example, the character class [aeiou] matches any lower case vowel, while [ˆaeiou] matches any character that is not a lower case vowel. Note that a circumflex is just a convenient notation for specifying the characters that are in the class by enumerating those that are not. A class that starts with a circumflex is not an assertion: it still consumes a character from the subject string, and therefore it fails if the current pointer is at the end of the string.

In UTF-8 mode, characters with values greater than 255 can be included in a class as a literal string of bytes, or by using the \x{ escaping mechanism.

When caseless matching is set, any letters in a class represent both their upper case and lower case versions, so for example, a caseless [aeiou] matches "A" as well as "a", and a caseless [ˆaeiou] does not match "A", whereas a caseful version would. In UTF-8 mode, PCRE always understands the concept of case for characters whose values are less than 128, so caseless matching is always possible. For characters with higher values, the concept of case is supported if PCRE is compiled with Unicode property support, but not otherwise. If you want to use caseless matching for characters 128 and above, you must ensure that PCRE is compiled with Unicode property support as well as with UTF-8 support.

Characters that might indicate line breaks are never treated in any special way when matching character classes, whatever line-ending sequence is in use, and whatever setting of the PCRE_DOTALL and PCRE_MULTILINE options is used. A class such as [ˆa] always matches one of these characters.

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, [d-m] matches any letter between d and m, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class.

It is not possible to have the literal character "]" as the end character of a range. A pattern such as [W-]46] is interpreted as a class of two characters ("W" and "-") followed by a literal string "46]", so it would match "W46]" or "-46]". However, if the "]" is escaped with a backslash it is interpreted as the end of range, so [W-\]46] is interpreted as a class containing a range followed by two other characters. The octal or hexadecimal representation of "]" can also be used to end a range.

Ranges operate in the collating sequence of character values. They can also be used for characters specified numerically, for example [\000-\037]. In UTF-8 mode, ranges can include characters whose values are greater than 255, for example [\x{100}-\x{2ff}].

If a range that includes letters is used when caseless matching is set, it matches the letters in either case. For example, [W-c] is equivalent to [][\\ˆ_‘wxyzabc], matched caselessly, and in non-UTF-8 mode, if character tables for the "fr_FR" locale are in use, [\xc8-\xcb] matches accented E characters in both cases. In UTF-8 mode, PCRE supports the concept of case for characters with values greater than 128 only when it is compiled with Unicode property support.

The character types \d, \D, \p, \P, \s, \S, \w, and \W may also appear in a character class, and add the characters that they match to the class. For example, [\dABCDEF] matches any hexadecimal digit. A circumflex can conveniently be used with the upper case character types to specify a more restricted set of characters than the matching lower case type. For example, the class [^\W_] matches any letter or digit, but not underscore.

The only metacharacters that are recognized in character classes are backslash, hyphen (only where it can be interpreted as specifying a range), circumflex (only at the start), opening square bracket (only when it can be interpreted as introducing a POSIX class name - see the next section), and the terminating closing square bracket. However, escaping other non-alphanumeric characters does no harm.

POSIX CHARACTER CLASSES

Perl supports the POSIX notation for character classes. This uses names enclosed by [: and :] within the enclosing square brackets. PCRE also supports this notation. For example,

```
  [01[:alpha:]%]
```

matches "0", "1", any alphabetic character, or "%". The supported class names are

alnum	letters and digits
alpha	letters
ascii	character codes 0 - 127
blank	space or tab only
cntrl	control characters
digit	decimal digits (same as \d)
graph	printing characters, excluding space
lower	lower case letters
print	printing characters, including space
punct	printing characters, excluding letters and digits
space	white space (not quite the same as \s)
upper	upper case letters
word	"word" characters (same as \w)
xdigit	hexadecimal digits

The "space" characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32). Notice that this list includes the VT character (code 11). This makes "space" different to \s, which does not include VT (for Perl compatibility).

The name "word" is a Perl extension, and "blank" is a GNU extension from Perl 5.8. Another Perl extension is negation, which is indicated by a ^ character after the colon. For example,

```
  [12[:^digit:]]
```

matches "1", "2", or any non-digit. PCRE (and Perl) also recognize the POSIX syntax [.ch.] and [=ch=] where "ch" is a "collating element", but these are not supported, and an error is given if they are encountered.

In UTF-8 mode, characters with values greater than 128 do not match any of the POSIX character classes.

VERTICAL BAR

Vertical bar characters are used to separate alternative patterns. For example, the pattern

```
  gilbert|sullivan
```

matches either "gilbert" or "sullivan". Any number of alternatives may appear, and an empty alternative is permitted (matching the empty string). The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. If the alternatives are within a subpattern (defined below), "succeeds" means matching the rest of the main pattern as well as the alternative in the subpattern.

INTERNAL OPTION SETTING

The settings of the PCRE_CASELESS, PCRE_MULTILINE, PCRE_DOTALL, and PCRE_EXTENDED options can be changed from within the pattern by a sequence of Perl option letters enclosed between "(?" and ")". The option letters are

```
  i  for PCRE_CASELESS
  m  for PCRE_MULTILINE
  s  for PCRE_DOTALL
  x  for PCRE_EXTENDED
```

For example, (?im) sets caseless, multiline matching. It is also possible to unset these options by preceding the letter with a hyphen, and a combined setting and unsetting such as (?im-sx), which sets PCRE_CASELESS and PCRE_MULTILINE while unsetting PCRE_DOTALL and PCRE_EXTENDED, is also permitted. If a letter appears both before and after the hyphen, the option is unset.

When an option change occurs at top level (that is, not inside subpattern parentheses), the change applies to the remainder of the pattern that follows. If the change is placed right at the start of a pattern, PCRE extracts it into the global options (and it will therefore show up in data extracted by the **pcre_fullinfo()** function).

An option change within a subpattern (see below for a description of subpatterns) affects only that part of the current pattern that follows it, so

```
  (a(?i)b)c
```

matches abc and aBc and no other strings (assuming PCRE_CASELESS is not used). By this means, options can be made to have different settings in different parts of the pattern. Any changes made in one alternative do carry on into subsequent branches within the same subpattern. For example,

```
  (a(?i)b|c)
```

matches "ab", "aB", "c", and "C", even though when matching "C" the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time. There would be some very weird behaviour otherwise.

The PCRE-specific options PCRE_DUPNAMES, PCRE_UNGREEDY, and PCRE_EXTRA can be changed in the same way as the Perl-compatible options by using the characters J, U and X respectively.

SUBPATTERNS

Subpatterns are delimited by parentheses (round brackets), which can be nested. Turning part of a pattern into a subpattern does two things:

1. It localizes a set of alternatives. For example, the pattern

```
  cat(aract|erpillar|)
```

matches one of the words "cat", "cataract", or "caterpillar". Without the parentheses, it would match "cataract", "erpillar" or an empty string.

2. It sets up the subpattern as a capturing subpattern. This means that, when the whole pattern matches, that portion of the subject string that matched the subpattern is passed back to the caller via the *ovector* argument of **pcre_exec()**. Opening parentheses are counted from left to right (starting from 1) to obtain numbers for the capturing subpatterns.

For example, if the string "the red king" is matched against the pattern

```
  the ((red|white) (king|queen))
```

the captured substrings are "red king", "red", and "king", and are numbered 1, 2, and 3, respectively.

The fact that plain parentheses fulfil two functions is not always helpful. There are often times when a grouping subpattern is required without a capturing requirement. If an opening parenthesis is followed by a question mark and a colon, the subpattern does not do any capturing, and is not counted when computing the number of any subsequent capturing subpatterns. For example, if the string "the white queen" is matched against the pattern

```
  the ((?:red|white) (king|queen))
```

the captured substrings are "white queen" and "queen", and are numbered 1 and 2. The maximum number of capturing subpatterns is 65535.

As a convenient shorthand, if any option settings are required at the start of a non-capturing subpattern, the option letters may appear between the "?" and the ":". Thus the two patterns

```
  (?i:saturday|sunday)
  (?:(?i)saturday|sunday)
```

match exactly the same set of strings. Because alternative branches are tried from left to right, and options are not reset until the end of the subpattern is reached, an option setting in one branch does affect subsequent branches, so the above patterns match "SUNDAY" as well as "Saturday".

NAMED SUBPATTERNS

Identifying capturing parentheses by number is simple, but it can be very hard to keep track of the numbers in complicated regular expressions. Furthermore, if an expression is modified, the numbers may change. To help with this difficulty, PCRE supports the naming of subpatterns. This feature was not added to Perl until release 5.10. Python had the feature earlier, and PCRE introduced it at release 4.0, using the Python syntax. PCRE now supports both the Perl and the Python syntax.

In PCRE, a subpattern can be named in one of three ways: (?<name>...) or (?'name'...) as in Perl, or (?P<name>...) as in Python. References to capturing parentheses from other parts of the pattern, such as backreferences, recursion, and conditions, can be made by name as well as by number.

Names consist of up to 32 alphanumeric characters and underscores. Named capturing parentheses are still allocated numbers as well as names, exactly as if the names were not present. The PCRE API provides function calls for extracting the name-to-number translation table from a compiled pattern. There is also a convenience function for extracting a captured substring by name.

By default, a name must be unique within a pattern, but it is possible to relax this constraint by setting the PCRE_DUPNAMES option at compile time. This can be useful for patterns where only one instance of the named parentheses can match. Suppose you want to match the name of a weekday, either as a 3-letter abbreviation or as the full name, and in both cases you want to extract the abbreviation. This pattern (ignoring the line breaks) does the job:

```
  (?<DN>Mon|Fri|Sun)(?:day)?|
  (?<DN>Tue)(?:sday)?|
  (?<DN>Wed)(?:nesday)?|
  (?<DN>Thu)(?:rsday)?|
  (?<DN>Sat)(?:urday)?
```

There are five capturing substrings, but only one is ever set after a match. The convenience function for extracting the data by name returns the substring for the first (and in this example, the only) subpattern of that name that matched. This saves searching to find which numbered subpattern it was. If you make a reference to a non-unique named subpattern from elsewhere in the pattern, the one that corresponds to the lowest number is used. For further details of the interfaces for handling named subpatterns, see the **pcreapi** documentation.

REPETITION

Repetition is specified by quantifiers, which can follow any of the following items:

```
  a literal data character
  the dot metacharacter
  the \C escape sequence
  the \X escape sequence (in UTF-8 mode with Unicode properties)
  the \R escape sequence
  an escape such as \d that matches a single character
  a character class
  a back reference (see next section)
  a parenthesized subpattern (unless it is an assertion)
```

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in curly brackets (braces), separated by a comma. The numbers must be less than 65536, and the first must be less than or equal to the second. For example:

```
  z{2,4}
```

matches "zz", "zzz", or "zzzz". A closing brace on its own is not a special character. If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus

```
  [aeiou]{3,}
```

matches at least 3 successive vowels, but may match many more, while

```
  \d{8}
```

matches exactly 8 digits. An opening curly bracket that appears in a position where a quantifier is not allowed, or one that does not match the syntax of a quantifier, is taken as a literal character. For example, {,6} is not a quantifier, but a literal string of four characters.

In UTF-8 mode, quantifiers apply to UTF-8 characters rather than to individual bytes. Thus, for example, \x{100}{2} matches two UTF-8 characters, each of which is represented by a two-byte sequence. Similarly, when Unicode property support is available, \X{3} matches three Unicode extended sequences, each of which may be several bytes long (and they may be of different lengths).

The quantifier {0} is permitted, causing the expression to behave as if the previous item and the quantifier were not present.

For convenience, the three most common quantifiers have single-character abbreviations:

```
  *    is equivalent to {0,}
  +    is equivalent to {1,}
  ?    is equivalent to {0,1}
```

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example:

```
  (a?)*
```

Earlier versions of Perl and PCRE used to give an error at compile time for such patterns. However, because there are cases where this can be useful, such patterns are now accepted, but if any repetition of the subpattern does in fact match no characters, the loop is forcibly broken.

By default, the quantifiers are "greedy", that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between /* and */ and within the comment, individual * and / characters may appear. An attempt to match C comments by applying the pattern

```
  /\*.*\*/
```

to the string

```
  /* first comment */  not comment  /* second comment */
```

fails, because it matches the entire string owing to the greediness of the .* item.

However, if a quantifier is followed by a question mark, it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern

```
  /\*.*?\*/
```

does the right thing with the C comments. The meaning of the various quantifiers is not otherwise changed, just the preferred number of matches. Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in

```
  \d??\d
```

which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

If the PCRE_UNGREEDY option is set (an option that is not available in Perl), the quantifiers are not greedy by default, but individual ones can be made greedy by following them with a question mark. In other words, it inverts the default behaviour.

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more memory is required for the compiled pattern, in proportion to the size of the minimum or maximum.

If a pattern starts with .* or .{0,} and the PCRE_DOTALL option (equivalent to Perl's /s) is set, thus allowing the dot to match newlines, the pattern is implicitly anchored, because whatever follows will be tried against every character position in the subject string, so there is no point in retrying the overall match at any position after the first. PCRE normally treats such a pattern as though it were preceded by \A.

In cases where it is known that the subject string contains no newlines, it is worth setting PCRE_DOTALL in order to obtain this optimization, or alternatively using ˆ to indicate anchoring explicitly.

However, there is one situation where the optimization cannot be used. When .* is inside capturing parentheses that are the subject of a backreference elsewhere in the pattern, a match at the start may fail where a later one succeeds. Consider, for example:

```
  (.*)abc\1
```

If the subject is "xyz123abc123" the match point is the fourth character. For this reason, such a pattern is not implicitly anchored.

When a capturing subpattern is repeated, the value captured is the substring that matched the final iteration. For example, after

```
  (tweedle[dume]{3}\s*)+
```

has matched "tweedledum tweedledee" the value of the captured substring is "tweedledee". However, if there are nested capturing subpatterns, the corresponding captured values may have been set in previous iterations. For example, after

```
  /(a|(b))+/
```

matches "aba" the value of the second captured substring is "b".

ATOMIC GROUPING AND POSSESSIVE QUANTIFIERS

With both maximizing ("greedy") and minimizing ("ungreedy" or "lazy") repetition, failure of what follows normally causes the repeated item to be re-evaluated to see if a different number of repeats allows the rest of the pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it fail earlier than it otherwise might, when the author of the pattern knows there is no point in carrying on.

Consider, for example, the pattern \d+foo when applied to the subject line

```
  123456bar
```

After matching all 6 digits and then failing to match "foo", the normal action of the matcher is to try again with only 5 digits matching the \d+ item, and then with 4, and so on, before ultimately failing. "Atomic grouping" (a term taken from Jeffrey Friedl's book) provides the means for specifying that once a subpattern has matched, it is not to be re-evaluated in this way.

If we use atomic grouping for the previous example, the matcher gives up immediately on failing to match "foo" the first time. The notation is a kind of special parenthesis, starting with (?> as in this example:

```
  (?>\d+)foo
```

This kind of parenthesis "locks up" the part of the pattern it contains once it has matched, and a failure further into the pattern is prevented from backtracking into it. Backtracking past it to previous items, however, works as normal.

An alternative description is that a subpattern of this type matches the string of characters that an identical standalone pattern would match, if anchored at the current point in the subject string.

Atomic grouping subpatterns are not capturing subpatterns. Simple cases such as the above example can be thought of as a maximizing repeat that must swallow everything it can. So, while both \d+ and \d+? are prepared to adjust the number of digits they match in order to make the rest of the pattern match, (?>\d+) can only match an entire sequence of digits.

Atomic groups in general can of course contain arbitrarily complicated subpatterns, and can be nested. However, when the subpattern for an atomic group is just a single repeated item, as in the example above, a simpler notation, called a "possessive quantifier" can be used. This consists of an additional + character following a quantifier. Using this notation, the previous example can be rewritten as

```
  \d++foo
```

Possessive quantifiers are always greedy; the setting of the PCRE_UNGREEDY option is ignored. They are a convenient notation for the simpler forms of atomic group. However, there is no difference in the meaning of a possessive quantifier and the equivalent atomic group, though there may be a performance difference; possessive quantifiers should be slightly faster.

The possessive quantifier syntax is an extension to the Perl 5.8 syntax. Jeffrey Friedl originated the idea (and the name) in the first edition of his book. Mike McCloskey liked it, so implemented it when he built Sun's Java package, and PCRE copied it from there. It ultimately found its way into Perl at release 5.10.

PCRE has an optimization that automatically "possessifies" certain simple pattern constructs. For example, the sequence A+B is treated as A++B because there is no point in backtracking into a sequence of A's when B must follow.

When a pattern contains an unlimited repeat inside a subpattern that can itself be repeated an unlimited number of times, the use of an atomic group is the only way to avoid some failing matches taking a very long time indeed. The pattern

```
  (\D+|<\d+>)*[!?]
```

matches an unlimited number of substrings that either consist of non-digits, or digits enclosed in <>, followed by either ! or ?. When it matches, it runs quickly. However, if it is applied to

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

it takes a long time before reporting failure. This is because the string can be divided between the internal \D+ repeat and the external * repeat in a large number of ways, and all have to be tried. (The example uses [!?] rather than a single character at the end, because both PCRE and Perl have an optimization that allows for fast failure when a single character is used. They remember the last single character that is required for a match, and fail early if it is not present in the string.) If the pattern is changed so that it uses an atomic group, like this:

```
((?>\D+)|<\d+>)*[!?]
```

sequences of non-digits cannot be broken, and failure happens quickly.

BACK REFERENCES

Outside a character class, a backslash followed by a digit greater than 0 (and possibly further digits) is a back reference to a capturing subpattern earlier (that is, to its left) in the pattern, provided there have been that many previous capturing left parentheses.

However, if the decimal number following the backslash is less than 10, it is always taken as a back reference, and causes an error only if there are not that many capturing left parentheses in the entire pattern. In other words, the parentheses that are referenced need not be to the left of the reference for numbers less than 10. A "forward back reference" of this type can make sense when a repetition is involved and the subpattern to the right has participated in an earlier iteration.

It is not possible to have a numerical "forward back reference" to a subpattern whose number is 10 or more using this syntax because a sequence such as \50 is interpreted as a character defined in octal. See the subsection entitled "Non-printing characters" above for further details of the handling of digits following a backslash. There is no such problem when named parentheses are used. A back reference to any subpattern is possible using named parentheses (see below).

Another way of avoiding the ambiguity inherent in the use of digits following a backslash is to use the \g escape sequence, which is a feature introduced in Perl 5.10. This escape must be followed by a positive or a negative number, optionally enclosed in braces. These examples are all identical:

```
(ring), \1
(ring), \g1
(ring), \g{1}
```

A positive number specifies an absolute reference without the ambiguity that is present in the older syntax. It is also useful when literal digits follow the reference. A negative number is a relative reference. Consider this example:

```
(abc(def)ghi)\g{-1}
```

The sequence \g{-1} is a reference to the most recently started capturing subpattern before \g, that is, is it equivalent to \2. Similarly, \g{-2} would be equivalent to \1. The use of relative references can be helpful in long patterns, and also in patterns that are created by joining together fragments that contain references within themselves.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself (see "Subpatterns as subroutines" below for a way of doing that). So the pattern

```
(sens|respons)e and \1ibility
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If caseful matching is in force at the time of the back reference, the case of letters is relevant. For example,

```
((?i)rah)\s+\1
```

matches "rah rah" and "RAH RAH", but not "RAH rah", even though the original capturing subpattern is matched caselessly.

Back references to named subpatterns use the Perl syntax \k<name> or \k’name’ or the Python syntax (?P=name). We could rewrite the above example in either of the following ways:

```
(?<p1>(?i)rah)\s+\k<p1>
(?P<p1>(?i)rah)\s+(?P=p1)
```

A subpattern that is referenced by name may appear in the pattern before or after the reference.

There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, any back references to it always fail. For example, the pattern

```
(a|(bc))\2
```

always fails if it starts to match "a" rather than "bc". Because there may be many capturing parentheses in a pattern, all digits following the backslash are taken as part of a potential back reference number. If the pattern continues with a digit character, some delimiter must be used to terminate the back reference. If the PCRE_EXTENDED option is set, this can be whitespace. Otherwise an empty comment (see "Comments" below) can be used.

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, (a\1) never matches. However, such references can be useful inside repeated subpatterns. For example, the pattern

```
(a|b\1)+
```

matches any number of "a"s and also "aba", "ababbaa" etc. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such that the first iteration does not need to match the back reference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.

ASSERTIONS

An assertion is a test on the characters following or preceding the current matching point that does not actually consume any characters. The simple assertions coded as \b, \B, \A, \G, \Z, \z, ^ and $ are described above.

More complicated assertions are coded as subpatterns. There are two kinds: those that look ahead of the current position in the subject string, and those that look behind it. An assertion subpattern is matched in the normal way, except that it does not cause the current matching position to be changed.

Assertion subpatterns are not capturing subpatterns, and may not be repeated, because it makes no sense to assert the same thing several times. If any kind of assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions, because it does not make sense for negative assertions.

Lookahead assertions

Lookahead assertions start with (?= for positive assertions and (?! for negative assertions. For example,

```
\w+(?=;)
```

matches a word followed by a semicolon, but does not include the semicolon in the match, and

```
foo(?!bar)
```

matches any occurrence of "foo" that is not followed by "bar". Note that the apparently similar pattern

```
  (?!foo)bar
```

does not find an occurrence of "bar" that is preceded by something other than "foo"; it finds any occurrence of "bar" whatsoever, because the assertion (?!foo) is always true when the next three characters are "bar". A lookbehind assertion is needed to achieve the other effect.

If you want to force a matching failure at some point in a pattern, the most convenient way to do it is with (?!) because an empty string always matches, so an assertion that requires there not to be an empty string must always fail.

Lookbehind assertions

Lookbehind assertions start with (?<= for positive assertions and (?<! for negative assertions. For example,

```
  (?<!foo)bar
```

does find an occurrence of "bar" that is not preceded by "foo". The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are several top-level alternatives, they do not all have to have the same fixed length. Thus

```
  (?<=bullock|donkey)
```

is permitted, but

```
  (?<!dogs?|cats?)
```

causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. This is an extension compared with Perl (at least for 5.8), which requires all branches to match the same length of string. An assertion such as

```
  (?<=ab(c|de))
```

is not permitted, because its single top-level branch can match two different lengths, but it is acceptable if rewritten to use two top-level branches:

```
  (?<=abc|abde)
```

The implementation of lookbehind assertions is, for each alternative, to temporarily move the current position back by the fixed length and then try to match. If there are insufficient characters before the current position, the assertion fails.

PCRE does not allow the \C escape (which matches a single byte in UTF-8 mode) to appear in lookbehind assertions, because it makes it impossible to calculate the length of the lookbehind. The \X and \R escapes, which can match different numbers of bytes, are also not permitted.

Possessive quantifiers can be used in conjunction with lookbehind assertions to specify efficient matching at the end of the subject string. Consider a simple pattern such as

```
  abcd$
```

when applied to a long string that does not match. Because matching proceeds from left to right, PCRE will look for each "a" in the subject and then see if what follows matches the rest of the pattern. If the pattern is specified as

```
  ^.*abcd$
```

the initial .* matches the entire string at first, but when this fails (because there is no following "a"), it backtracks to match all but the last character, then all but the last two characters, and so on. Once again the search for "a" covers the entire string, from right to left, so we are no better off. However, if the pattern is written as

```
^.*+(?<=abcd)
```

there can be no backtracking for the .*+ item; it can match only the entire string. The subsequent look-behind assertion does a single test on the last four characters. If it fails, the match fails immediately. For long strings, this approach makes a significant difference to the processing time.

Using multiple assertions

Several assertions (of any sort) may occur in succession. For example,

```
(?<=\d{3})(?<!999)foo
```

matches "foo" preceded by three digits that are not "999". Notice that each of the assertions is applied independently at the same point in the subject string. First there is a check that the previous three characters are all digits, and then there is a check that the same three characters are not "999". This pattern does *not* match "foo" preceded by six characters, the first of which are digits and the last three of which are not "999". For example, it doesn't match "123abcfoo". A pattern to do that is

```
(?<=\d{3}...)(?<!999)foo
```

This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not "999".

Assertions can be nested in any combination. For example,

```
(?<=(?<!foo)bar)baz
```

matches an occurrence of "baz" that is preceded by "bar" which in turn is not preceded by "foo", while

```
(?<=\d{3}(?!999)...)foo
```

is another pattern that matches "foo" preceded by three digits and any three characters that are not "999".

CONDITIONAL SUBPATTERNS

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a previous capturing subpattern matched or not. The two possible forms of conditional subpattern are

```
(?(condition)yes-pattern)
(?(condition)yes-pattern|no-pattern)
```

If the condition is satisfied, the yes-pattern is used; otherwise the no-pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile-time error occurs.

There are four kinds of condition: references to subpatterns, references to recursion, a pseudo-condition called DEFINE, and assertions.

Checking for a used subpattern by number

If the text between the parentheses consists of a sequence of digits, the condition is true if the capturing subpattern of that number has previously matched.

Consider the following pattern, which contains non-significant white space to make it more readable (assume the PCRE_EXTENDED option) and to divide it into three parts for ease of discussion:

```
( \( )?    [^()]+    (?(1) \) )
```

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether the first set of parentheses matched or not. If they did, that is, if subject started with an opening parenthesis, the condition is true, and so the yes-pattern is executed and a closing parenthesis is required. Otherwise, since no-pattern is not present, the subpattern matches nothing. In other words, this pattern matches a sequence of non-parentheses, optionally enclosed in parentheses.

Checking for a used subpattern by name

Perl uses the syntax (?(<name>)...) or (?('name')...) to test for a used subpattern by name. For compatibility with earlier versions of PCRE, which had this facility before Perl, the syntax (?(name)...) is also recognized. However, there is a possible ambiguity with this syntax, because subpattern names may consist entirely of digits. PCRE looks first for a named subpattern; if it cannot find one and the name consists entirely of digits, PCRE looks for a subpattern of that number, which must be greater than zero. Using subpattern names that consist entirely of digits is not recommended.

Rewriting the above example to use a named subpattern gives this:

```
(?<OPEN> \( )?    [^()]+    (?(<OPEN>) \) )
```

Checking for pattern recursion

If the condition is the string (R), and there is no subpattern with the name R, the condition is true if a recursive call to the whole pattern or any subpattern has been made. If digits or a name preceded by ampersand follow the letter R, for example:

```
(?(R3)...) or (?(R&name)...)
```

the condition is true if the most recent recursion is into the subpattern whose number or name is given. This condition does not check the entire recursion stack.

At "top level", all these recursion test conditions are false. Recursive patterns are described below.

Defining subpatterns for use by reference only

If the condition is the string (DEFINE), and there is no subpattern with the name DEFINE, the condition is always false. In this case, there may be only one alternative in the subpattern. It is always skipped if control reaches this point in the pattern; the idea of DEFINE is that it can be used to define "subroutines" that can be referenced from elsewhere. (The use of "subroutines" is described below.) For example, a pattern to match an IPv4 address could be written like this (ignore whitespace and line breaks):

```
(?(DEFINE) (?<byte> 2[0-4]\d | 25[0-5] | 1\d\d | [1-9]?\d) )
\b (?&byte) (\.(?&byte)){3} \b
```

The first part of the pattern is a DEFINE group inside which a another group named "byte" is defined. This matches an individual component of an IPv4 address (a number less than 256). When matching takes place, this part of the pattern is skipped because DEFINE acts like a false condition.

The rest of the pattern uses references to the named group to match the four dot-separated components of an IPv4 address, insisting on a word boundary at each end.

Assertion conditions

If the condition is not in any of the above formats, it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern, again containing non-significant white space, and with the two alternatives on the second line:

```
(?(?=[^a-z]*[a-z])
\d{2}-[a-z]{3}-\d{2}  |  \d{2}-\d{2}-\d{2} )
```

The condition is a positive lookahead assertion that matches an optional sequence of non-letters followed by a letter. In other words, it tests for the presence of at least one letter in the subject. If a letter is found, the subject is matched against the first alternative; otherwise it is matched against the second.

This pattern matches strings in one of the two forms dd-aaa-dd or dd-dd-dd, where aaa are letters and dd are digits.

COMMENTS

The sequence (?# marks the start of a comment that continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part in the pattern matching at all.

If the PCRE_EXTENDED option is set, an unescaped # character outside a character class introduces a comment that continues to immediately after the next newline in the pattern.

RECURSIVE PATTERNS

Consider the problem of matching a string in parentheses, allowing for unlimited nested parentheses. Without the use of recursion, the best that can be done is to use a pattern that matches up to some fixed depth of nesting. It is not possible to handle an arbitrary nesting depth.

For some time, Perl has provided a facility that allows regular expressions to recurse (amongst other things). It does this by interpolating Perl code in the expression at run time, and the code can refer to the expression itself. A Perl pattern using code interpolation to solve the parentheses problem can be created like this:

```
  $re = qr{\( (?: (?>[^()]+) | (?p{$re}) )* \)}x;
```

The (?p{...}) item interpolates Perl code at run time, and in this case refers recursively to the pattern in which it appears.

Obviously, PCRE cannot support the interpolation of Perl code. Instead, it supports special syntax for recursion of the entire pattern, and also for individual subpattern recursion. After its introduction in PCRE and Python, this kind of recursion was introduced into Perl at release 5.10.

A special item that consists of (? followed by a number greater than zero and a closing parenthesis is a recursive call of the subpattern of the given number, provided that it occurs inside that subpattern. (If not, it is a "subroutine" call, which is described in the next section.) The special item (?R) or (?0) is a recursive call of the entire regular expression.

In PCRE (like Python, but unlike Perl), a recursive subpattern call is always treated as an atomic group. That is, once it has matched some of the subject string, it is never re-entered, even if it contains untried alternatives and there is a subsequent matching failure.

This PCRE pattern solves the nested parentheses problem (assume the PCRE_EXTENDED option is set so that white space is ignored):

```
  \( ( (?>[^()]+) | (?R) )* \)
```

First it matches an opening parenthesis. Then it matches any number of substrings which can either be a sequence of non-parentheses, or a recursive match of the pattern itself (that is, a correctly parenthesized substring). Finally there is a closing parenthesis.

If this were part of a larger pattern, you would not want to recurse the entire pattern, so instead you could use this:

```
  ( \( ( (?>[^()]+) | (?1) )* \) )
```

We have put the pattern into parentheses, and caused the recursion to refer to them instead of the whole pattern. In a larger pattern, keeping track of parenthesis numbers can be tricky. It may be more convenient to use named parentheses instead. The Perl syntax for this is (?&name); PCRE's earlier syntax (?P>name) is also supported. We could rewrite the above example as follows:

```
  (?<pn> \( ( (?>[^()]+) | (?&pn) )* \) )
```

If there is more than one subpattern with the same name, the earliest one is used. This particular example pattern contains nested unlimited repeats, and so the use of atomic grouping for matching strings of non-parentheses is important when applying the pattern to strings that do not match. For example, when this pattern is applied to

```
  (aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa()
```

it yields "no match" quickly. However, if atomic grouping is not used, the match runs for a very long time indeed because there are so many different ways the + and * repeats can carve up the subject, and all have to be tested before failure can be reported.

At the end of a match, the values set for any capturing subpatterns are those from the outermost level of the recursion at which the subpattern value is set. If you want to obtain intermediate values, a callout function can be used (see below and the **pcrecallout** documentation). If the pattern above is matched against

```
  (ab(cd)ef)
```

the value for the capturing parentheses is "ef", which is the last value taken on at the top level. If additional parentheses are added, giving

```
  \( ( ( (?>[^()]+) | (?R) )* ) \)
     ^                   ^
     ^                   ^
```

the string they capture is "ab(cd)ef", the contents of the top level parentheses. If there are more than 15 capturing parentheses in a pattern, PCRE has to obtain extra memory to store data during a recursion, which it does by using **pcre_malloc**, freeing it via **pcre_free** afterwards. If no memory can be obtained, the match fails with the PCRE_ERROR_NOMEMORY error.

Do not confuse the (?R) item with the condition (R), which tests for recursion. Consider this pattern, which matches text in angle brackets, allowing for arbitrary nesting. Only digits are allowed in nested brackets (that is, when recursing), whereas any characters are permitted at the outer level.

```
  < (?: (?(R) \d++  | [^<>]*+) | (?R)) * >
```

In this pattern, (?(R) is the start of a conditional subpattern, with two different alternatives for the recursive and non-recursive cases. The (?R) item is the actual recursive call.

SUBPATTERNS AS SUBROUTINES

If the syntax for a recursive subpattern reference (either by number or by name) is used outside the parentheses to which it refers, it operates like a subroutine in a programming language. The "called" subpattern may be defined before or after the reference. An earlier example pointed out that the pattern

```
  (sens|respons)e and \1ibility
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If instead the pattern

```
  (sens|respons)e and (?1)ibility
```

is used, it does match "sense and responsibility" as well as the other two strings. Another example is given in the discussion of DEFINE above.

Like recursive subpatterns, a "subroutine" call is always treated as an atomic group. That is, once it has matched some of the subject string, it is never re-entered, even if it contains untried alternatives and there is a subsequent matching failure.

When a subpattern is used as a subroutine, processing options such as case-independence are fixed when the subpattern is defined. They cannot be changed for different calls. For example, consider this pattern:

```
  (abc)(?i:(?1))
```

It matches "abcabc". It does not match "abcABC" because the change of processing option does not affect the called subpattern.

CALLOUTS

Perl has a feature whereby using the sequence (?{...}) causes arbitrary Perl code to be obeyed in the middle of matching a regular expression. This makes it possible, amongst other things, to extract different substrings that match the same pair of parentheses when there is a repetition.

PCRE provides a similar feature, but of course it cannot obey arbitrary Perl code. The feature is called "callout". The caller of PCRE provides an external function by putting its entry point in the global variable *pcre_callout*. By default, this variable contains NULL, which disables all calling out.

Within a regular expression, (?C) indicates the points at which the external function is to be called. If you want to identify different callout points, you can put a number less than 256 after the letter C. The default value is zero. For example, this pattern has two callout points:

```
(?C1)abc(?C2)def
```

If the PCRE_AUTO_CALLOUT flag is passed to **pcre_compile()**, callouts are automatically installed before each item in the pattern. They are all numbered 255.

During matching, when PCRE reaches a callout point (and *pcre_callout* is set), the external function is called. It is provided with the number of the callout, the position in the pattern, and, optionally, one item of data originally supplied by the caller of **pcre_exec()**. The callout function may cause matching to proceed, to backtrack, or to fail altogether. A complete description of the interface to the callout function is given in the **pcrecallout** documentation.

SEE ALSO

pcreapi(3), **pcrecallout**(3), **pcrematching**(3), **pcre**(3).

Last updated: 06 December 2006

NAME

PCRE - Perl-compatible regular expressions

PCRE PERFORMANCE

Two aspects of performance are discussed below: memory usage and processing time. The way you express your pattern as a regular expression can affect both of them.

MEMORY USAGE

Patterns are compiled by PCRE into a reasonably efficient byte code, so that most simple patterns do not use much memory. However, there is one case where memory usage can be unexpectedly large. When a parenthesized subpattern has a quantifier with a minimum greater than 1 and/or a limited maximum, the whole subpattern is repeated in the compiled code. For example, the pattern

```
  (abc|def){2,4}
```

is compiled as if it were

```
  (abc|def)(abc|def)((abc|def)(abc|def)?)?
```

(Technical aside: It is done this way so that backtrack points within each of the repetitions can be independently maintained.)

For regular expressions whose quantifiers use only small numbers, this is not usually a problem. However, if the numbers are large, and particularly if such repetitions are nested, the memory usage can become an embarrassment. For example, the very simple pattern

```
  ((ab){1,1000}c){1,3}
```

uses 51K bytes when compiled. When PCRE is compiled with its default internal pointer size of two bytes, the size limit on a compiled pattern is 64K, and this is reached with the above pattern if the outer repetition is increased from 3 to 4. PCRE can be compiled to use larger internal pointers and thus handle larger compiled patterns, but it is better to try to rewrite your pattern to use less memory if you can.

One way of reducing the memory usage for such patterns is to make use of PCRE's "subroutine" facility. Re-writing the above pattern as

```
  ((ab)(?2){0,999}c)(?1){0,2}
```

reduces the memory requirements to 18K, and indeed it remains under 20K even with the outer repetition increased to 100. However, this pattern is not exactly equivalent, because the "subroutine" calls are treated as atomic groups into which there can be no backtracking if there is a subsequent matching failure. Therefore, PCRE cannot do this kind of rewriting automatically. Furthermore, there is a noticeable loss of speed when executing the modified pattern. Nevertheless, if the atomic grouping is not a problem and the loss of speed is acceptable, this kind of rewriting will allow you to process patterns that PCRE cannot otherwise handle.

PROCESSING TIME

Certain items in regular expression patterns are processed more efficiently than others. It is more efficient to use a character class like [aeiou] than a set of single-character alternatives such as (a|e|i|o|u). In general, the simplest construction that provides the required behaviour is usually the most efficient. Jeffrey Friedl's book contains a lot of useful general discussion about optimizing regular expressions for efficient performance. This document contains a few observations about PCRE.

Using Unicode character properties (the \p, \P, and \X escapes) is slow, because PCRE has to scan a structure that contains data for over fifteen thousand characters whenever it needs a character's property. If you can find an alternative pattern that does not use character properties, it will probably be faster.

When a pattern begins with .* not in parentheses, or in parentheses that are not the subject of a backreference, and the PCRE_DOTALL option is set, the pattern is implicitly anchored by PCRE, since it can match only at the start of a subject string. However, if PCRE_DOTALL is not set, PCRE cannot make this optimization, because the . metacharacter does not then match a newline, and if the subject string contains newlines, the pattern may match from the character immediately following one of them instead of from the very start. For example, the pattern

```
.*second
```

matches the subject "first\nand second" (where \n stands for a newline character), with the match starting at the seventh character. In order to do this, PCRE has to retry the match starting after every newline in the subject.

If you are using such a pattern with subject strings that do not contain newlines, the best performance is obtained by setting PCRE_DOTALL, or starting the pattern with ˆ.* or ˆ.*? to indicate explicit anchoring. That saves PCRE from having to scan along the subject looking for a newline to restart at.

Beware of patterns that contain nested indefinite repeats. These can take a long time to run when applied to a string that does not match. Consider the pattern fragment

```
ˆ(a+)*
```

This can match "aaaa" in 16 different ways, and this number increases very rapidly as the string gets longer. (The * repeat can match 0, 1, 2, 3, or 4 times, and for each of those cases other than 0 or 4, the + repeats can match different numbers of times.) When the remainder of the pattern is such that the entire match is going to fail, PCRE has in principle to try every possible variation, and this can take an extremely long time, even for relatively short strings.

An optimization catches some of the more simple cases such as

```
(a+)*b
```

where a literal character follows. Before embarking on the standard matching procedure, PCRE checks that there is a "b" later in the subject string, and if there is not, it fails the match immediately. However, when there is no following literal this optimization cannot be used. You can see the difference by comparing the behaviour of

```
(a+)*\d
```

with the pattern above. The former gives a failure almost instantly when applied to a whole line of "a" characters, whereas the latter takes an appreciable time with strings longer than about 20 characters.

In many cases, the solution to this kind of performance issue is to use an atomic group or a possessive quantifier.

Last updated: 20 September 2006

NAME

PCRE - Perl-compatible regular expressions.

SYNOPSIS OF POSIX API

#include <pcreposix.h>

int regcomp(regex_t ****preg*, const char ****pattern***,**
int *cflags***);**

int regexec(regex_t ****preg*, const char ****string***,**
size_t *nmatch***, regmatch_t** *pmatch***[], int** *eflags***);**

size_t regerror(int *errcode***, const regex_t ****preg***,**
char ****errbuf*, size_t** *errbuf_size***);**

void regfree(regex_t ****preg*);**

DESCRIPTION

This set of functions provides a POSIX-style API to the PCRE regular expression package. See the **pcreapi** documentation for a description of PCRE's native API, which contains much additional functionality.

The functions described here are just wrapper functions that ultimately call the PCRE native API. Their prototypes are defined in the **pcreposix.h** header file, and on Unix systems the library itself is called **pcreposix.a**, so can be accessed by adding **-lpcreposix** to the command for linking an application that uses them. Because the POSIX functions call the native ones, it is also necessary to add **-lpcre**.

I have implemented only those option bits that can be reasonably mapped to PCRE native options. In addition, the option REG_EXTENDED is defined with the value zero. This has no effect, but since programs that are written to the POSIX interface often use it, this makes it easier to slot in PCRE as a replacement library. Other POSIX options are not even defined.

When PCRE is called via these functions, it is only the API that is POSIX-like in style. The syntax and semantics of the regular expressions themselves are still those of Perl, subject to the setting of various PCRE options, as described below. "POSIX-like in style" means that the API approximates to the POSIX definition; it is not fully POSIX-compatible, and in multi-byte encoding domains it is probably even less compatible.

The header for these functions is supplied as **pcreposix.h** to avoid any potential clash with other POSIX libraries. It can, of course, be renamed or aliased as **regex.h**, which is the "correct" name. It provides two structure types, *regex_t* for compiled internal forms, and *regmatch_t* for returning captured substrings. It also defines some constants whose names start with "REG_"; these are used for setting options and identifying error codes.

COMPILING A PATTERN

The function **regcomp()** is called to compile a pattern into an internal form. The pattern is a C string terminated by a binary zero, and is passed in the argument *pattern*. The *preg* argument is a pointer to a **regex_t** structure that is used as a base for storing information about the compiled regular expression.

The argument *cflags* is either zero, or contains one or more of the bits defined by the following macros:

```
  REG_DOTALL
```

The PCRE_DOTALL option is set when the regular expression is passed for compilation to the native function. Note that REG_DOTALL is not part of the POSIX standard.

```
  REG_ICASE
```

The PCRE_CASELESS option is set when the regular expression is passed for compilation to the native function.

```
  REG_NEWLINE
```

The PCRE_MULTILINE option is set when the regular expression is passed for compilation to the native function. Note that this does *not* mimic the defined POSIX behaviour for REG_NEWLINE (see the following section).

```
  REG_NOSUB
```

The PCRE_NO_AUTO_CAPTURE option is set when the regular expression is passed for compilation to the native function. In addition, when a pattern that is compiled with this flag is passed to **regexec()** for matching, the *nmatch* and *pmatch* arguments are ignored, and no captured strings are returned.

```
  REG_UTF8
```

The PCRE_UTF8 option is set when the regular expression is passed for compilation to the native function. This causes the pattern itself and all data strings used for matching it to be treated as UTF-8 strings. Note that REG_UTF8 is not part of the POSIX standard.

In the absence of these flags, no options are passed to the native function. This means the the regex is compiled with PCRE default semantics. In particular, the way it handles newline characters in the subject string is the Perl way, not the POSIX way. Note that setting PCRE_MULTILINE has only *some* of the effects specified for REG_NEWLINE. It does not affect the way newlines are matched by . (they aren't) or by a negative class such as [^a] (they are).

The yield of **regcomp()** is zero on success, and non-zero otherwise. The *preg* structure is filled in on success, and one member of the structure is public: *re_nsub* contains the number of capturing subpatterns in the regular expression. Various error codes are defined in the header file.

MATCHING NEWLINE CHARACTERS

This area is not simple, because POSIX and Perl take different views of things. It is not possible to get PCRE to obey POSIX semantics, but then PCRE was never intended to be a POSIX engine. The following table lists the different possibilities for matching newline characters in PCRE:

```
                        Default   Change with

  . matches newline         no     PCRE_DOTALL
  newline matches [^a]       yes    not changeable
  $ matches \n at end       yes    PCRE_DOLLARENDONLY
  $ matches \n in middle     no     PCRE_MULTILINE
  ^ matches \n in middle     no     PCRE_MULTILINE
```

This is the equivalent table for POSIX:

```
                        Default   Change with

  . matches newline         yes    REG_NEWLINE
  newline matches [^a]       yes    REG_NEWLINE
  $ matches \n at end       no     REG_NEWLINE
  $ matches \n in middle     no     REG_NEWLINE
  ^ matches \n in middle     no     REG_NEWLINE
```

PCRE's behaviour is the same as Perl's, except that there is no equivalent for PCRE_DOLLAR_ENDONLY in Perl. In both PCRE and Perl, there is no way to stop newline from matching [^a].

The default POSIX newline handling can be obtained by setting PCRE_DOTALL and PCRE_DOLLAR_ENDONLY, but there is no way to make PCRE behave exactly as for the REG_NEWLINE action.

MATCHING A PATTERN

The function **regexec()** is called to match a compiled pattern *preg* against a given *string*, which is terminated by a zero byte, subject to the options in *eflags*. These can be:

```
  REG_NOTBOL
```

The PCRE_NOTBOL option is set when calling the underlying PCRE matching function.

```
  REG_NOTEOL
```

The PCRE_NOTEOL option is set when calling the underlying PCRE matching function.

If the pattern was compiled with the REG_NOSUB flag, no data about any matched strings is returned. The *nmatch* and *pmatch* arguments of **regexec()** are ignored.

Otherwise,the portion of the string that was matched, and also any captured substrings, are returned via the *pmatch* argument, which points to an array of *nmatch* structures of type *regmatch_t*, containing the members *rm_so* and *rm_eo*. These contain the offset to the first character of each substring and the offset to the first character after the end of each substring, respectively. The 0th element of the vector relates to the entire portion of *string* that was matched; subsequent elements relate to the capturing subpatterns of the regular expression. Unused entries in the array have both structure members set to -1.

A successful match yields a zero return; various error codes are defined in the header file, of which REG_NOMATCH is the "expected" failure code.

ERROR MESSAGES

The **regerror()** function maps a non-zero errorcode from either **regcomp()** or **regexec()** to a printable message. If *preg* is not NULL, the error should have arisen from the use of that structure. A message terminated by a binary zero is placed in *errbuf*. The length of the message, including the zero, is limited to *errbuf_size*. The yield of the function is the size of buffer needed to hold the whole message.

MEMORY USAGE

Compiling a regular expression causes memory to be allocated and associated with the *preg* structure. The function **regfree()** frees all such memory, after which *preg* may no longer be used as a compiled expression.

AUTHOR

Philip Hazel
University Computing Service,
Cambridge CB2 3QH, England.

Last updated: 16 January 2006

NAME

PCRE - Perl-compatible regular expressions

SAVING AND RE-USING PRECOMPILED PCRE PATTERNS

If you are running an application that uses a large number of regular expression patterns, it may be useful to store them in a precompiled form instead of having to compile them every time the application is run. If you are not using any private character tables (see the **pcre_maketables()** documentation), this is relatively straightforward. If you are using private tables, it is a little bit more complicated.

If you save compiled patterns to a file, you can copy them to a different host and run them there. This works even if the new host has the opposite endianness to the one on which the patterns were compiled. There may be a small performance penalty, but it should be insignificant.

SAVING A COMPILED PATTERN

The value returned by **pcre_compile()** points to a single block of memory that holds the compiled pattern and associated data. You can find the length of this block in bytes by calling **pcre_fullinfo()** with an argument of PCRE_INFO_SIZE. You can then save the data in any appropriate manner. Here is sample code that compiles a pattern and writes it to a file. It assumes that the variable *fd* refers to a file that is open for output:

```
  int erroroffset, rc, size;
  char *error;
  pcre *re;

  re = pcre_compile("my pattern", 0, &error, &erroroffset, NULL);
  if (re == NULL) { ... handle errors ... }
  rc = pcre_fullinfo(re, NULL, PCRE_INFO_SIZE, &size);
  if (rc < 0) { ... handle errors ... }
  rc = fwrite(re, 1, size, fd);
  if (rc != size) { ... handle errors ... }
```

In this example, the bytes that comprise the compiled pattern are copied exactly. Note that this is binary data that may contain any of the 256 possible byte values. On systems that make a distinction between binary and non-binary data, be sure that the file is opened for binary output.

If you want to write more than one pattern to a file, you will have to devise a way of separating them. For binary data, preceding each pattern with its length is probably the most straightforward approach. Another possibility is to write out the data in hexadecimal instead of binary, one pattern to a line.

Saving compiled patterns in a file is only one possible way of storing them for later use. They could equally well be saved in a database, or in the memory of some daemon process that passes them via sockets to the processes that want them.

If the pattern has been studied, it is also possible to save the study data in a similar way to the compiled pattern itself. When studying generates additional information, **pcre_study()** returns a pointer to a **pcre_extra** data block. Its format is defined in the section on matching a pattern in the **pcreapi** documentation. The *study_data* field points to the binary study data, and this is what you must save (not the **pcre_extra** block itself). The length of the study data can be obtained by calling **pcre_fullinfo()** with an argument of PCRE_INFO_STUDYSIZE. Remember to check that **pcre_study()** did return a non-NULL value before trying to save the study data.

RE-USING A PRECOMPILED PATTERN

Re-using a precompiled pattern is straightforward. Having reloaded it into main memory, you pass its pointer to **pcre_exec()** or **pcre_dfa_exec()** in the usual way. This should work even on another host, and even if that host has the opposite endianness to the one where the pattern was compiled.

However, if you passed a pointer to custom character tables when the pattern was compiled (the *tableptr* argument of **pcre_compile()**), you must now pass a similar pointer to **pcre_exec()** or **pcre_dfa_exec()**, because the value saved with the compiled pattern will obviously be nonsense. A field in a **pcre_extra()** block is used to pass this data, as described in the section on matching a pattern in the **pcreapi** documentation.

If you did not provide custom character tables when the pattern was compiled, the pointer in the compiled pattern is NULL, which causes **pcre_exec()** to use PCRE's internal tables. Thus, you do not need to take any special action at run time in this case.

If you saved study data with the compiled pattern, you need to create your own **pcre_extra** data block and set the *study_data* field to point to the reloaded study data. You must also set the PCRE_EXTRA_STUDY_DATA bit in the *flags* field to indicate that study data is present. Then pass the **pcre_extra** block to **pcre_exec()** or **pcre_dfa_exec()** in the usual way.

COMPATIBILITY WITH DIFFERENT PCRE RELEASES

The layout of the control block that is at the start of the data that makes up a compiled pattern was changed for release 5.0. If you have any saved patterns that were compiled with previous releases (not a facility that was previously advertised), you will have to recompile them for release 5.0 and above.

If you have any saved patterns in UTF-8 mode that use \p or \P that were compiled with any release up to and including 6.4, you will have to recompile them for release 6.5 and above.

All saved patterns from earlier releases must be recompiled for release 7.0 or higher, because there was an internal reorganization at that release.

Last updated: 28 November 2006

NAME

PCRE - Perl-compatible regular expressions

PCRE SAMPLE PROGRAM

A simple, complete demonstration program, to get you started with using PCRE, is supplied in the file *pcredemo.c* in the PCRE distribution.

The program compiles the regular expression that is its first argument, and matches it against the subject string in its second argument. No PCRE options are set, and default character tables are used. If matching succeeds, the program outputs the portion of the subject that matched, together with the contents of any captured substrings.

If the -g option is given on the command line, the program then goes on to check for further matches of the same regular expression in the same subject string. The logic is a little bit tricky because of the possibility of matching an empty string. Comments in the code explain what is going on.

If PCRE is installed in the standard include and library directories for your system, you should be able to compile the demonstration program using this command:

```
gcc -o pcredemo pcredemo.c -lpcre
```

If PCRE is installed elsewhere, you may need to add additional options to the command line. For example, on a Unix-like system that has PCRE installed in */usr/local*, you can compile the demonstration program using a command like this:

```
gcc -o pcredemo -I/usr/local/include pcredemo.c \
    -L/usr/local/lib -lpcre
```

Once you have compiled the demonstration program, you can run simple tests like this:

```
./pcredemo 'cat|dog' 'the cat sat on the mat'
./pcredemo -g 'cat|dog' 'the dog sat on the cat'
```

Note that there is a much more comprehensive test program, called **pcretest**, which supports many more facilities for testing regular expressions and the PCRE library. The **pcredemo** program is provided as a simple coding example.

On some operating systems (e.g. Solaris), when PCRE is not installed in the standard library directory, you may get an error like this when you try to run **pcredemo**:

```
ld.so.1: a.out: fatal: libpcre.so.0: open failed: No such file or directory
```

This is caused by the way shared library support works on those systems. You need to add

```
-R/usr/local/lib
```

(for example) to the compile command to get round this problem.

Last updated: 09 September 2004

NAME

PCRE - Perl-compatible regular expressions

PCRE DISCUSSION OF STACK USAGE

When you call **pcre_exec()**, it makes use of an internal function called **match()**. This calls itself recursively at branch points in the pattern, in order to remember the state of the match so that it can back up and try a different alternative if the first one fails. As matching proceeds deeper and deeper into the tree of possibilities, the recursion depth increases.

Not all calls of **match()** increase the recursion depth; for an item such as a* it may be called several times at the same level, after matching different numbers of a's. Furthermore, in a number of cases where the result of the recursive call would immediately be passed back as the result of the current call (a "tail recursion"), the function is just restarted instead.

The **pcre_dfa_exec()** function operates in an entirely different way, and hardly uses recursion at all. The limit on its complexity is the amount of workspace it is given. The comments that follow do NOT apply to **pcre_dfa_exec()**; they are relevant only for **pcre_exec()**.

You can set limits on the number of times that **match()** is called, both in total and recursively. If the limit is exceeded, an error occurs. For details, see the section on extra data for **pcre_exec()** in the **pcreapi** documentation.

Each time that **match()** is actually called recursively, it uses memory from the process stack. For certain kinds of pattern and data, very large amounts of stack may be needed, despite the recognition of "tail recursion". You can often reduce the amount of recursion, and therefore the amount of stack used, by modifying the pattern that is being matched. Consider, for example, this pattern:

```
  ([^<]|<(?!inet))+
```

It matches from wherever it starts until it encounters "<inet" or the end of the data, and is the kind of pattern that might be used when processing an XML file. Each iteration of the outer parentheses matches either one character that is not "<" or a "<" that is not followed by "inet". However, each time a parenthesis is processed, a recursion occurs, so this formulation uses a stack frame for each matched character. For a long string, a lot of stack is required. Consider now this rewritten pattern, which matches exactly the same strings:

```
  ([^<]++|<(?!inet))
```

This uses very much less stack, because runs of characters that do not contain "<" are "swallowed" in one item inside the parentheses. Recursion happens only when a "<" character that is not followed by "inet" is encountered (and we assume this is relatively rare). A possessive quantifier is used to stop any backtracking into the runs of non-"<" characters, but that is not related to stack usage.

This example shows that one way of avoiding stack problems when matching long subject strings is to write repeated parenthesized subpatterns to match more than one character whenever possible.

In environments where stack memory is constrained, you might want to compile PCRE to use heap memory instead of stack for remembering back-up points. This makes it run a lot more slowly, however. Details of how to do this are given in the **pcrebuild** documentation.

In Unix-like environments, there is not often a problem with the stack unless very long strings are involved, though the default limit on stack size varies from system to system. Values from 8Mb to 64Mb are common. You can find your default limit by running the command:

```
  ulimit -s
```

Unfortunately, the effect of running out of stack is often SIGSEGV, though sometimes a more explicit error message is given. You can normally increase the limit on stack size by code such as this:

```
  struct rlimit rlim;
  getrlimit(RLIMIT_STACK, &rlim);
  rlim.rlim_cur = 100*1024*1024;
  setrlimit(RLIMIT_STACK, &rlim);
```

This reads the current limits (soft and hard) using **getrlimit()**, then attempts to increase the soft limit to 100Mb using **setrlimit()**. You must do this before calling **pcre_exec()**.

PCRE has an internal counter that can be used to limit the depth of recursion, and thus cause **pcre_exec()** to give an error code before it runs out of stack. By default, the limit is very large, and unlikely ever to operate. It can be changed when PCRE is built, and it can also be set when **pcre_exec()** is called. For details of these interfaces, see the **pcrebuild** and **pcreapi** documentation.

As a very rough rule of thumb, you should reckon on about 500 bytes per recursion. Thus, if you want to limit your stack usage to 8Mb, you should set the limit at 16000 recursions. A 64Mb stack, on the other hand, can support around 128000 recursions. The **pcretest** test program has a command line option (**-S**) that can be used to increase the size of its stack.

Last updated: 14 September 2006

NAME

PCRE - Perl-compatible regular expressions

SYNOPSIS

#include <pcre.h>

pcre *pcre_compile(const char ****pattern*, int** *options***,**
  const char *****errptr*, int ****erroffset***,**
  const unsigned char ****tableptr*);**

DESCRIPTION

This function compiles a regular expression into an internal form. It is the same as **pcre_compile2()**, except for the absence of the *errorcodeptr* argument. Its arguments are:

```
  pattern     A zero-terminated string containing the
                regular expression to be compiled
  options     Zero or more option bits
  errptr      Where to put an error message
  erroffset   Offset in pattern where error was found
  tableptr    Pointer to character tables, or NULL to
                use the built-in default
```

The option bits are:

```
  PCRE_ANCHORED         Force pattern anchoring
  PCRE_AUTO_CALLOUT     Compile automatic callouts
  PCRE_CASELESS         Do caseless matching
  PCRE_DOLLAR_ENDONLY   $ not to match newline at end
  PCRE_DOTALL           . matches anything including NL
  PCRE_DUPNAMES         Allow duplicate names for subpatterns
  PCRE_EXTENDED         Ignore whitespace and # comments
  PCRE_EXTRA            PCRE extra features
                          (not much use currently)
  PCRE_FIRSTLINE        Force matching to be before newline
  PCRE_MULTILINE        ^ and $ match newlines within data
  PCRE_NEWLINE_ANY      Recognize any Unicode newline sequence
  PCRE_NEWLINE_CR       Set CR as the newline sequence
  PCRE_NEWLINE_CRLF     Set CRLF as the newline sequence
  PCRE_NEWLINE_LF       Set LF as the newline sequence
  PCRE_NO_AUTO_CAPTURE  Disable numbered capturing paren-
                          theses (named ones available)
  PCRE_UNGREEDY         Invert greediness of quantifiers
  PCRE_UTF8             Run in UTF-8 mode
  PCRE_NO_UTF8_CHECK    Do not check the pattern for UTF-8
                          validity (only relevant if
                          PCRE_UTF8 is set)
```

PCRE must be built with UTF-8 support in order to use PCRE_UTF8 and PCRE_NO_UTF8_CHECK.

The yield of the function is a pointer to a private data structure that contains the compiled pattern, or NULL if an error was detected.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

NAME

PCRE - Perl-compatible regular expressions

SYNOPSIS

#include <pcre.h>

pcre *pcre_compile2(const char ***_pattern_, int** _options_**,**
int ***_errorcodeptr_,**
const char ****_errptr_, int ***_erroffset_**,**
const unsigned char ***_tableptr_);**

DESCRIPTION

This function compiles a regular expression into an internal form. It is the same as **pcre_compile()**, except for the addition of the _errorcodeptr_ argument. The arguments are:

```
  pattern       A zero-terminated string containing the
                  regular expression to be compiled
  options       Zero or more option bits
  errorcodeptr  Where to put an error code
  errptr        Where to put an error message
  erroffset     Offset in pattern where error was found
  tableptr      Pointer to character tables, or NULL to
                  use the built-in default
```

The option bits are:

```
  PCRE_ANCHORED         Force pattern anchoring
  PCRE_AUTO_CALLOUT     Compile automatic callouts
  PCRE_CASELESS         Do caseless matching
  PCRE_DOLLAR_ENDONLY   $ not to match newline at end
  PCRE_DOTALL           . matches anything including NL
  PCRE_DUPNAMES         Allow duplicate names for subpatterns
  PCRE_EXTENDED         Ignore whitespace and # comments
  PCRE_EXTRA            PCRE extra features
                          (not much use currently)
  PCRE_FIRSTLINE        Force matching to be before newline
  PCRE_MULTILINE        ^ and $ match newlines within data
  PCRE_NEWLINE_ANY      Recognize any Unicode newline sequence
  PCRE_NEWLINE_CR       Set CR as the newline sequence
  PCRE_NEWLINE_CRLF     Set CRLF as the newline sequence
  PCRE_NEWLINE_LF       Set LF as the newline sequence
  PCRE_NO_AUTO_CAPTURE  Disable numbered capturing paren-
                          theses (named ones available)
  PCRE_UNGREEDY         Invert greediness of quantifiers
  PCRE_UTF8             Run in UTF-8 mode
  PCRE_NO_UTF8_CHECK    Do not check the pattern for UTF-8
                          validity (only relevant if
                          PCRE_UTF8 is set)
```

PCRE must be built with UTF-8 support in order to use PCRE_UTF8 and PCRE_NO_UTF8_CHECK.

The yield of the function is a pointer to a private data structure that contains the compiled pattern, or NULL if an error was detected.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

NAME

PCRE - Perl-compatible regular expressions

SYNOPSIS

#include <pcre.h>

int pcre_config(int *what***, void ****where***);**

DESCRIPTION

This function makes it possible for a client program to find out which optional features are available in the version of the PCRE library it is using. Its arguments are as follows:

```
  what     A code specifying what information is required
  where    Points to where to put the data
```

The available codes are:

```
  PCRE_CONFIG_LINK_SIZE     Internal link size: 2, 3, or 4
  PCRE_CONFIG_MATCH_LIMIT   Internal resource limit
  PCRE_CONFIG_MATCH_LIMIT_RECURSION
                            Internal recursion depth limit
  PCRE_CONFIG_NEWLINE       Value of the newline sequence:
                                13 (0x000d)    for CR
                                10 (0x000a)    for LF
                              3338 (0x0d0a)    for CRLF
                                -1             for ANY
  PCRE_CONFIG_POSIX_MALLOC_THRESHOLD
                            Threshold of return slots, above
                              which malloc() is used by
                              the POSIX API
  PCRE_CONFIG_STACKRECURSE  Recursion implementation (1=stack 0=heap)
  PCRE_CONFIG_UTF8          Availability of UTF-8 support (1=yes 0=no)
  PCRE_CONFIG_UNICODE_PROPERTIES
                            Availability of Unicode property support
                              (1=yes 0=no)
```

The function yields 0 on success or PCRE_ERROR_BADOPTION otherwise.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

NAME

PCRE - Perl-compatible regular expressions

SYNOPSIS

#include <pcre.h>

int pcre_copy_named_substring(const pcre ***_code_,**
  const char ***_subject_, int ***_ovector_**,**
  int _stringcount_**, const char ***_stringname_**,**
  char ***_buffer_, int** _buffersize_**);**

DESCRIPTION

This is a convenience function for extracting a captured substring, identified by name, into a given buffer. The arguments are:

code	Pattern that was successfully matched
subject	Subject that has been successfully matched
ovector	Offset vector that **pcre_exec()** used
stringcount	Value returned by **pcre_exec()**
stringname	Name of the required substring
buffer	Buffer to receive the string
buffersize	Size of buffer

The yield is the length of the substring, PCRE_ERROR_NOMEMORY if the buffer was too small, or PCRE_ERROR_NOSUBSTRING if the string name is invalid.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

NAME

PCRE - Perl-compatible regular expressions

SYNOPSIS

#include <pcre.h>

int pcre_copy_substring(const char ****subject*, int ****ovector***,**
int *stringcount***, int** *stringnumber***, char ****buffer***,**
int *buffersize***);**

DESCRIPTION

This is a convenience function for extracting a captured substring into a given buffer. The arguments are:

subject	Subject that has been successfully matched
ovector	Offset vector that **pcre_exec()** used
stringcount	Value returned by **pcre_exec()**
stringnumber	Number of the required substring
buffer	Buffer to receive the string
buffersize	Size of buffer

The yield is the length of the string, PCRE_ERROR_NOMEMORY if the buffer was too small, or PCRE_ERROR_NOSUBSTRING if the string number is invalid.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

NAME

PCRE - Perl-compatible regular expressions

SYNOPSIS

#include <pcre.h>

**int pcre_dfa_exec(const pcre ****code*, **const pcre_extra ****extra*,
    const char ****subject*, **int *length*, **int** *startoffset*,
    int *options*, **int ****ovector*, **int** *ovecsize*,
    int ****workspace*, **int *wscount***);**

DESCRIPTION

This function matches a compiled regular expression against a given subject string, using an alternative matching algorithm that scans the subject string just once (*not* Perl-compatible). Note that the main, Perl-compatible, matching function is **pcre_exec()**. The arguments for this function are:

code	Points to the compiled pattern
extra	Points to an associated **pcre_extra** structure, or is NULL
subject	Points to the subject string
length	Length of the subject string, in bytes
startoffset	Offset in bytes in the subject at which to start matching
options	Option bits
ovector	Points to a vector of ints for result offsets
ovecsize	Number of elements in the vector
workspace	Points to a vector of ints used as working space
wscount	Number of elements in the vector

The options are:

PCRE_ANCHORED	Match only at the first position
PCRE_NEWLINE_ANY	Recognize any Unicode newline sequence
PCRE_NEWLINE_CR	Set CR as the newline sequence
PCRE_NEWLINE_CRLF	Set CRLF as the newline sequence
PCRE_NEWLINE_LF	Set LF as the newline sequence
PCRE_NOTBOL	Subject is not the beginning of a line
PCRE_NOTEOL	Subject is not the end of a line
PCRE_NOTEMPTY	An empty string is not a valid match
PCRE_NO_UTF8_CHECK	Do not check the subject for UTF-8 validity (only relevant if PCRE_UTF8 was set at compile time)
PCRE_PARTIAL	Return PCRE_ERROR_PARTIAL for a partial match
PCRE_DFA_SHORTEST	Return only the shortest match
PCRE_DFA_RESTART	This is a restart after a partial match

There are restrictions on what may appear in a pattern when using this matching function. Details are given in the **pcrematching** documentation.

A **pcre_extra** structure contains the following fields:

flags	Bits indicating which fields are set
study_data	Opaque data from **pcre_study()**
match_limit	Limit on internal resource use
match_limit_recursion	Limit on internal recursion depth
callout_data	Opaque data passed back to callouts
tables	Points to character tables or is NULL

The flag bits are PCRE_EXTRA_STUDY_DATA, PCRE_EXTRA_MATCH_LIMIT, PCRE_EXTRA_MATCH_LIMIT_RECURSION, PCRE_EXTRA_CALLOUT_DATA, and PCRE_EXTRA_TABLES. For this matching function, the *match_limit* and *match_limit_recursion* fields are not used, and must not be set.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

NAME

PCRE - Perl-compatible regular expressions

SYNOPSIS

#include <pcre.h>

**int pcre_exec(const pcre ****code*, **const pcre_extra ****extra*,
const char ****subject*, **int *length*, **int** *startoffset*,
int *options*, **int ****ovector*, **int** *ovecsize***);**

DESCRIPTION

This function matches a compiled regular expression against a given subject string, using a matching algorithm that is similar to Perl's. It returns offsets to captured substrings. Its arguments are:

code	Points to the compiled pattern
extra	Points to an associated **pcre_extra** structure, or is NULL
subject	Points to the subject string
length	Length of the subject string, in bytes
startoffset	Offset in bytes in the subject at which to start matching
options	Option bits
ovector	Points to a vector of ints for result offsets
ovecsize	Number of elements in the vector (a multiple of 3)

The options are:

PCRE_ANCHORED	Match only at the first position
PCRE_NEWLINE_ANY	Recognize any Unicode newline sequence
PCRE_NEWLINE_CR	Set CR as the newline sequence
PCRE_NEWLINE_CRLF	Set CRLF as the newline sequence
PCRE_NEWLINE_LF	Set LF as the newline sequence
PCRE_NOTBOL	Subject is not the beginning of a line
PCRE_NOTEOL	Subject is not the end of a line
PCRE_NOTEMPTY	An empty string is not a valid match
PCRE_NO_UTF8_CHECK	Do not check the subject for UTF-8 validity (only relevant if PCRE_UTF8 was set at compile time)
PCRE_PARTIAL	Return PCRE_ERROR_PARTIAL for a partial match

There are restrictions on what may appear in a pattern when partial matching is requested. For details, see the **pcrepartial** page.

A **pcre_extra** structure contains the following fields:

flags	Bits indicating which fields are set
study_data	Opaque data from **pcre_study()**
match_limit	Limit on internal resource use
match_limit_recursion	Limit on internal recursion depth
callout_data	Opaque data passed back to callouts
tables	Points to character tables or is NULL

The flag bits are PCRE_EXTRA_STUDY_DATA, PCRE_EXTRA_MATCH_LIMIT, PCRE_EXTRA_MATCH_LIMIT_RECURSION, PCRE_EXTRA_CALLOUT_DATA, and PCRE_EXTRA_TABLES.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

NAME

PCRE - Perl-compatible regular expressions

SYNOPSIS

#include <pcre.h>

void pcre_free_substring(const char ****stringptr*);**

DESCRIPTION

This is a convenience function for freeing the store obtained by a previous call to **pcre_get_substring()** or **pcre_get_named_substring()**. Its only argument is a pointer to the string.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

NAME

PCRE - Perl-compatible regular expressions

SYNOPSIS

#include <pcre.h>

void pcre_free_substring_list(const char *****stringptr*);**

DESCRIPTION

This is a convenience function for freeing the store obtained by a previous call to **pcre_get_substring_list()**. Its only argument is a pointer to the list of string pointers.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

NAME

PCRE - Perl-compatible regular expressions

SYNOPSIS

#include <pcre.h>

**int pcre_fullinfo(const pcre ****code*, **const pcre_extra ****extra*,
int *what*, **void ****where***);**

DESCRIPTION

This function returns information about a compiled pattern. Its arguments are:

code	Compiled regular expression
extra	Result of **pcre_study()** or NULL
what	What information is required
where	Where to put the information

The following information is available:

```
  PCRE_INFO_BACKREFMAX      Number of highest back reference
  PCRE_INFO_CAPTURECOUNT    Number of capturing subpatterns
  PCRE_INFO_DEFAULT_TABLES  Pointer to default tables
  PCRE_INFO_FIRSTBYTE       Fixed first byte for a match, or
                          -1 for start of string
                             or after newline, or
                          -2 otherwise
  PCRE_INFO_FIRSTTABLE      Table of first bytes
                          (after studying)
  PCRE_INFO_LASTLITERAL     Literal last byte required
  PCRE_INFO_NAMECOUNT       Number of named subpatterns
  PCRE_INFO_NAMEENTRYSIZE   Size of name table entry
  PCRE_INFO_NAMETABLE       Pointer to name table
  PCRE_INFO_OPTIONS         Option bits used for compilation
  PCRE_INFO_SIZE            Size of compiled pattern
  PCRE_INFO_STUDYSIZE       Size of study data
```

The yield of the function is zero on success or:

```
  PCRE_ERROR_NULL           the argument code was NULL
                          the argument where was NULL
  PCRE_ERROR_BADMAGIC       the "magic number" was not found
  PCRE_ERROR_BADOPTION      the value of what was invalid
```

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

NAME

PCRE - Perl-compatible regular expressions

SYNOPSIS

#include <pcre.h>

int pcre_get_named_substring(const pcre ***code*,**
const char ***subject*, int ***ovector***,**
int *stringcount***, const char ***stringname***,**
const char ****stringptr*);**

DESCRIPTION

This is a convenience function for extracting a captured substring by name. The arguments are:

code	Compiled pattern
subject	Subject that has been successfully matched
ovector	Offset vector that **pcre_exec()** used
stringcount	Value returned by **pcre_exec()**
stringname	Name of the required substring
stringptr	Where to put the string pointer

The memory in which the substring is placed is obtained by calling **pcre_malloc()**. The convenience function **pcre_free_substring()** can be used to free it when it is no longer needed. The yield of the function is the length of the extracted substring, PCRE_ERROR_NOMEMORY if sufficient memory could not be obtained, or PCRE_ERROR_NOSUBSTRING if the string name is invalid.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

NAME

PCRE - Perl-compatible regular expressions

SYNOPSIS

#include <pcre.h>

int pcre_get_stringnumber(const pcre ****code*,**
const char ****name*);**

DESCRIPTION

This convenience function finds the number of a named substring capturing parenthesis in a compiled pattern. Its arguments are:

code	Compiled regular expression
name	Name whose number is required

The yield of the function is the number of the parenthesis if the name is found, or PCRE_ERROR_NOSUBSTRING otherwise. When duplicate names are allowed (PCRE_DUPNAMES is set), it is not defined which of the numbers is returned by **pcre_get_stringnumber()**. You can obtain the complete list by calling **pcre_get_stringtable_entries()**.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

NAME

PCRE - Perl-compatible regular expressions

SYNOPSIS

#include <pcre.h>

**int pcre_get_stringtable_entries(const pcre ****code*,
const char ****name*, **char *****first*, **char *****last*);**

DESCRIPTION

This convenience function finds, for a compiled pattern, the first and last entries for a given name in the table that translates capturing parenthesis names into numbers. When names are required to be unique (PCRE_DUPNAMES is *not* set), it is usually easier to use **pcre_get_stringnumber()** instead.

code	Compiled regular expression
name	Name whose entries required
first	Where to return a pointer to the first entry
last	Where to return a pointer to the last entry

The yield of the function is the length of each entry, or PCRE_ERROR_NOSUBSTRING if none are found.

There is a complete description of the PCRE native API, including the format of the table entries, in the **pcreapi** page, and a description of the POSIX API in the **pcreposix** page.

NAME

PCRE - Perl-compatible regular expressions

SYNOPSIS

#include <pcre.h>

int pcre_get_substring(const char ****subject*, **int **ovector*,
    int *stringcount*, **int** *stringnumber*,
    const char ***stringptr***);**

DESCRIPTION

This is a convenience function for extracting a captured substring. The arguments are:

subject	Subject that has been successfully matched
ovector	Offset vector that **pcre_exec()** used
stringcount	Value returned by **pcre_exec()**
stringnumber	Number of the required substring
stringptr	Where to put the string pointer

The memory in which the substring is placed is obtained by calling **pcre_malloc()**. The convenience function **pcre_free_substring()** can be used to free it when it is no longer needed. The yield of the function is the length of the substring, PCRE_ERROR_NOMEMORY if sufficient memory could not be obtained, or PCRE_ERROR_NOSUBSTRING if the string number is invalid.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

NAME

PCRE - Perl-compatible regular expressions

SYNOPSIS

#include <pcre.h>

int pcre_get_substring_list(const char ***_subject_,**
int ***_ovector_, int** _stringcount_**, const char *****_listptr_**);**

DESCRIPTION

This is a convenience function for extracting a list of all the captured substrings. The arguments are:

subject	Subject that has been successfully matched
ovector	Offset vector that **pcre_exec** used
stringcount	Value returned by **pcre_exec**
listptr	Where to put a pointer to the list

The memory in which the substrings and the list are placed is obtained by calling **pcre_malloc()**. The convenience function **pcre_free_substring_list()** can be used to free it when it is no longer needed. A pointer to a list of pointers is put in the variable whose address is in *listptr*. The list is terminated by a NULL pointer. The yield of the function is zero on success or PCRE_ERROR_NOMEMORY if sufficient memory could not be obtained.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

NAME

PCRE - Perl-compatible regular expressions

SYNOPSIS

#include <pcre.h>

int pcre_info(const pcre ****code*, int ****optptr***, int ****firstcharptr***);**

DESCRIPTION

This function is obsolete. You should be using **pcre_fullinfo()** instead.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

NAME

PCRE - Perl-compatible regular expressions

SYNOPSIS

#include <pcre.h>

const unsigned char *pcre_maketables(void);

DESCRIPTION

This function builds a set of character tables for character values less than 256. These can be passed to **pcre_compile()** to override PCRE's internal, built-in tables (which were made by **pcre_maketables()** when PCRE was compiled). You might want to do this if you are using a non-standard locale. The function yields a pointer to the tables.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

NAME

PCRE - Perl-compatible regular expressions

SYNOPSIS

#include <pcre.h>

int pcre_refcount(pcre ****code*, **int *adjust***);**

DESCRIPTION

This function is used to maintain a reference count inside a data block that contains a compiled pattern. Its arguments are:

code	Compiled regular expression
adjust	Adjustment to reference value

The yield of the function is the adjusted reference value, which is constrained to lie between 0 and 65535.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

NAME

PCRE - Perl-compatible regular expressions

SYNOPSIS

#include <pcre.h>

pcre_extra *pcre_study(const pcre ***_code_, int** _options_**,**
const char ****_errptr_);**

DESCRIPTION

This function studies a compiled pattern, to see if additional information can be extracted that might speed up matching. Its arguments are:

code	A compiled regular expression
options	Options for **pcre_study()**
errptr	Where to put an error message

If the function succeeds, it returns a value that can be passed to **pcre_exec()** via its *extra* argument.

If the function returns NULL, either it could not find any additional information, or there was an error. You can tell the difference by looking at the error value. It is NULL in first case.

There are currently no options defined; the value of the second argument should always be zero.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

NAME

PCRE - Perl-compatible regular expressions

SYNOPSIS

#include <pcre.h>

char *pcre_version(void);

DESCRIPTION

This function returns a character string that gives the version number of the PCRE library and the date of its release.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.