NAME

regcomp, regexec, regerror, regfree – regular expression matching

SYNOPSIS

#include <sys/types.h>
#include <regex.h>

int regcomp(regex_t **preg*, **const char** **pattern*, **int** *cflags***);**

int regexec(const 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***);**

MT-LEVEL

MT-Safe

DESCRIPTION

These functions interpret *basic* and *extended* regular expressions (described on the **regex**(5) manual page).

The structure type **regex_t** contains at least the following member:

size_t re_nsub	Number of parenthesised subexpressions.

The structure type **regmatch_t** contains at least the following members:

regoff_t rm_so	Byte offset from start of *string* to start of substring.
regoff_t rm_eo	Byte offset from start of *string* of the first character after the end of substring.

regcomp( )

The **regcomp( )** function will compile the regular expression contained in the string pointed to by the *pattern* argument and place the results in the structure pointed to by *preg*. The *cflags* argument is the bitwise inclusive OR of zero or more of the following flags, which are defined in the header **<regex.h>**:

REG_EXTENDED	Use Extended Regular Expressions.
REG_ICASE	Ignore case in match.
REG_NOSUB	Report only success/fail in **regexec( )**.
REG_NEWLINE	Change the handling of NEWLINE characters, as described in the text.

The default regular expression type for *pattern* is a Basic Regular Expression. The application can specify Extended Regular Expressions using the **REG_EXTENDED** *cflags* flag.

If the **REG_NOSUB** flag was not set in *cflags*, then **regcomp( )** will set *re_nsub* to the number of parenthesised subexpressions (delimited by \(\) in basic regular expressions or ( ) in extended regular expressions) found in *pattern*.

regexec( )

The **regexec( )** function compares the null-terminated string specified by *string* with the compiled regular expression *preg* initialized by a previous call to **regcomp( )**. The *eflags* argument is the bitwise inclusive OR of zero or more of the following flags, which are defined in the header **<regex.h>**:

REG_NOTBOL The first character of the string pointed to by *string* is not the beginning of the line. Therefore, the circumflex character (^), when taken as a special character, will not match the beginning of *string*.

REG_NOTEOL The last character of the string pointed to by *string* is not the end of the line. Therefore, the dollar sign ($), when taken as a special character, will not match the end of *string*.

If *nmatch* is zero or **REG_NOSUB** was set in the *cflags* argument to **regcomp( )**, then **regexec( )** will ignore the *pmatch* argument. Otherwise, the *pmatch* argument must point to an array with at least *nmatch* elements, and **regexec( )** will fill in the elements of that array with offsets of the substrings of *string* that correspond to the parenthesised subexpressions of *pattern*: *pmatch*[*i*].*rm_so* will be the byte offset of the beginning and *pmatch*[*i*].*rm_eo* will be one greater than the byte offset of the end of substring *i*. (Subexpression *i* begins at the *i*th matched open parenthesis, counting from 1.) Offsets in *pmatch*[**0**] identify the substring that corresponds to the entire regular expression. Unused elements of *pmatch* up to *pmatch*[*nmatch*–**1**] will be filled with –**1**. If there are more than *nmatch* subexpressions in *pattern* (*pattern* itself counts as a subexpression), then **regexec( )** will still do the match, but will record only the first *nmatch* substrings.

When matching a basic or extended regular expression, any given parenthesised subexpression of *pattern* might participate in the match of several different substrings of *string*, or it might not match any substring even though the pattern as a whole did match. The following rules are used to determine which substrings to report in *pmatch* when matching regular expressions:

1. If subexpression *i* in a regular expression is not contained within another subexpression, and it participated in the match several times, then the byte offsets in *pmatch*[*i*] will delimit the last such match.
2. If subexpression *i* is not contained within another subexpression, and it did not participate in an otherwise successful match, the byte offsets in *pmatch*[*i*] will be –**1**. A subexpression does not participate in the match when:

   * or \{ \} appears immediately after the subexpression in a basic regular expression, or *, **?**, or **{ }** appears immediately after the subexpression in an extended regular expression, and the subexpression did not match (matched zero times)

   or

   | is used in an extended regular expression to select this subexpression or another, and the other subexpression matched.
3. If subexpression *i* is contained within another subexpression *j*, and *i* is not contained within any other subexpression that is contained within *j*, and a match of subexpression *j* is reported in *pmatch*[*j*], then the match or non-match of subexpression *i* reported in *pmatch*[*i*] will be as described in 1. and 2. above, but within the substring reported in *pmatch*[*j*] rather than the whole string.
4. If subexpression *i* is contained in subexpression *j*, and the byte offsets in *pmatch*[*j*] are –**1**, then the pointers in *pmatch*[*i*] also will be –**1**.

5. If subexpression *i* matched a zero-length string, then both byte offsets in *pmatch*[*i*] will be the byte offset of the character or **NULL** terminator immediately following the zero-length string.

If, when **regexec()** is called, the locale is different from when the regular expression was compiled, the result is undefined.

If **REG_NEWLINE** is not set in *cflags*, then a NEWLINE character in *pattern* or *string* will be treated as an ordinary character. If **REG_NEWLINE** is set, then newline will be treated as an ordinary character except as follows:

1. A NEWLINE character in *string* will not be matched by a period outside a bracket expression or by any form of a non-matching list.
2. A circumflex (^) in *pattern*, when used to specify expression anchoring will match the zero-length string immediately after a newline in *string*, regardless of the setting of **REG_NOTBOL**.
3. A dollar-sign ($) in *pattern*, when used to specify expression anchoring, will match the zero-length string immediately before a newline in *string*, regardless of the setting of **REG_NOTEOL**.

regfree()

The **regfree()** function frees any memory allocated by **regcomp()** associated with *preg*.

The following constants are defined as error return values:

REG_NOMATCH	**regexec()** failed to match.
REG_BADPAT	Invalid regular expression.
REG_ECOLLATE	Invalid collating element referenced.
REG_ECTYPE	Invalid character class type referenced.
REG_EESCAPE	Trailing \ in pattern.
REG_ESUBREG	Number in *digit* invalid or in error.
REG_EBRACK	**[ ]** imbalance.
REG_ENOSYS	The function is not supported.
REG_EPAREN	**\\(\\)** or **()** imbalance.
REG_EBRACE	\\{ \\} imbalance.
REG_BADBR	Content of \\{ \\} invalid: not a number, number too large, more than two numbers, first larger than second.
REG_ERANGE	Invalid endpoint in range expression.
REG_ESPACE	Out of memory.
REG_BADRPT	?, * or + not preceded by valid regular expression.

regerror()

The **regerror()** function provides a mapping from error codes returned by **regcomp()** and **regexec()** to unspecified printable strings. It generates a string corresponding to the value of the *errcode* argument, which must be the last non-zero value returned by **regcomp()** or **regexec()** with the given value of *preg*. If *errcode* is not such a value, an error message indicating that the error code is invalid is returned.

If *preg* is a **NULL** pointer, but *errcode* is a value returned by a previous call to **regexec( )** or **regcomp( )**, the **regerror( )** still generates an error string corresponding to the value of *errcode*.

If the *errbuf_size* argument is not zero, **regerror( )** will place the generated string into the buffer of size *errbuf_size* bytes pointed to by *errbuf*. If the string (including the terminating **NULL**) cannot fit in the buffer, **regerror( )** will truncate the string and null-terminate the result.

If *errbuf_size* is zero, **regerror( )** ignores the *errbuf* argument, and returns the size of the buffer needed to hold the generated string.

If the *preg* argument to **regexec( )** or **regfree( )** is not a compiled regular expression returned by **regcomp( )**, the result is undefined. A *preg* is no longer treated as a compiled regular expression after it is given to **regfree( )**.

RETURN VALUES

The following values are returned by **regcomp( )**:

0 successful completion

non-zero an error has occurred. The value returned is described in **<regex.h>**, and the content of *preg* is undefined.

The following values are returned by **regexec( )**:

0 successful completion.

REG_NOMATCH no match

REG_ENOSYS the function is not supported.

The following values are returned by **regerror( )**:

0 the function is not implemented.

Upon successful completion, the function returns the number of bytes needed to hold the entire generated string.

The **regfree( )** function returns no value.

USAGE

An application could use:

regerror(code,preg,(char *)NULL,(size_t)0)

to find out how big a buffer is needed for the generated string, **malloc** a buffer to hold the string, and then call **regerror( )** again to get the string (see **malloc**(3C)). Alternately, it could allocate a fixed, static buffer that is big enough to hold most strings, and then use **malloc**( ) to allocate a larger buffer if it finds that this is too small.

EXAMPLES

```
#include <regex.h>

/*
 * Match string against the extended regular expression in
 * pattern, treating errors as no match.
 *
 * return 1 for match, 0 for no match
 */

int
match(const char *string, char *pattern)
{
	int	status;
	regex_t re;

	if (regcomp(&re, pattern, REG_EXTENDED | REG_NOSUB) != 0) {
		return(0);	/* report error */
	}
	status = regexec(&re, string, (size_t) 0, NULL, 0);
	regfree(&re);
	if (status != 0) {
		return(0);	/* report error */
	}
	return(1);
}
```

The following demonstrates how the **REG_NOTBOL** flag could be used with **regexec( )** to find all substrings in a line that match a pattern supplied by a user. (For simplicity of the example, very little error checking is done.)

```
(void) regcomp (&re, pattern, 0);
/* this call to regexec() finds the first match on the line */
error = regexec (&re, &buffer[0], 1, &pm, 0);
while (error == 0) {	/* while matches found */
	/* substring found between pm.rm_so and pm.rm_eo */
	/* This call to regexec() finds the next match */
	error = regexec (&re, buffer + pm.rm_eo, 1, &pm, REG_NOTBOL);
}
```

SEE ALSO

fnmatch(3C), **glob**(3C), **malloc**(3C), **regex**(5)