NAME

regcomp, regexec, regerror, regfree -- regular-expression library

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <regex.h>

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

int
    regexec(const regex_t * restrict preg,
            const char * restrict string,
            size_t nmatch,
            regmatch_t pmatch[restrict],
            int eflags);

size_t
    regerror(int errcode,
             const regex_t * restrict preg,
             char * restrict errbuf,
             size_t errbuf_size);

void
    regfree(regex_t *preg);

typedef struct
    {   int re_magic;
        size_t re_nsub;         /* number of parenthesized subexpressions */
        const char *re_endp;    /* end pointer for REG_PEND */
        struct re_guts *re_g;   /* none of your business :-) */
    } regex_t;

typedef struct
    {   regoff_t rm_so;         /* start of match */
        regoff_t rm_eo;         /* end of match */
    } regmatch_t;
```

DESCRIPTION

These routines implement IEEE Std 1003.2 (``POSIX.2'') regular expressions (``RE''s); see re_format(7). The regcomp() function compiles an RE written as a string into an internal form, regexec() matches that internal form against a string and reports results, regerror() transforms error codes from either into human-readable messages, and regfree() frees any dynamically-allocated storage used by the internal form of an RE.

The header <regex.h> declares two structure types, regex_t and regmatch_t, the former for compiled internal forms and the latter for match reporting. It also declares the four functions, a type regoff_t, and a number of constants with names starting with ``REG_''.

The regcomp() function compiles the regular expression contained in the pattern string, subject to the flags in cflags, and places the results in the regex_t structure pointed to by preg. The cflags argument is the bitwise OR of zero or more of the following flags:

REG_EXTENDED Compile modern (``extended'') REs, rather than the obsolete (``basic'') REs that are the default.

REG_BASIC This is a synonym for 0, provided as a counterpart to REG_EXTENDED to improve readability.

REG_NOSPEC Compile with recognition of all special characters turned off. All characters are thus considered ordinary, so the ``RE'' is a literal string. This is an extension, compatible with but not specified by IEEE Std 1003.2 (``POSIX.2''), and should be used with caution in software intended to be portable to other systems. REG_EXTENDED and REG_NOSPEC may not be used in the same call to regcomp().

REG_ICASE Compile for matching that ignores upper/lower case distinctions. See re_format(7).

REG_NOSUB Compile for matching that need only report success or failure, not what was matched.

REG_NEWLINE Compile for newline-sensitive matching. By default, newline is a completely ordinary character with no special meaning in either REs or strings. With this flag, `[^' bracket expressions and `.' never match newline, a `^' anchor matches the null string after any newline in the string in addition to its normal function, and the `$' anchor matches the null string before any newline in the string in addition to its normal function.

REG_PEND The regular expression ends, not at the first NUL, but just before the character pointed to by the re_endp member of the structure pointed to by preg. The re_endp member is of type const char *. This flag permits inclusion of NULs in the RE; they are considered ordinary characters. This is an extension, compatible with but not specified by IEEE Std 1003.2 (``POSIX.2''), and should be used with caution in software intended to be portable to other systems.

When successful, regcomp() returns 0 and fills in the structure pointed to by preg. One member of that structure (other than re_endp) is publicized: re_nsub, of type size_t, contains the number of parenthesized subexpressions within the RE (except that the value of this member is undefined if the REG_NOSUB flag was used). If regcomp() fails, it returns a non-zero error code; see DIAGNOSTICS.

The regexec() function matches the compiled RE pointed to by preg against the string, subject to the flags in eflags, and reports results using nmatch, pmatch, and the returned value. The RE must have been compiled by a previous invocation of regcomp(). The compiled form is not altered during execution of regexec(), so a single compiled RE can be used simultaneously by multiple threads.

By default, the NUL-terminated string pointed to by string is considered to be the text of an entire line, minus any terminating newline. The eflags argument is the bitwise OR of zero or more of the following flags:

REG_NOTBOL The first character of the string is not the beginning of a line, so the `^' anchor should not match before it. This does not affect the behavior of newlines under REG_NEWLINE.

REG_NOTEOL The NUL terminating the string does not end a line, so the `$' anchor should not match before it. This does not affect the behavior of newlines under REG_NEWLINE.

REG_STARTEND The string is considered to start at string + pmatch[0].rm_so and to have a terminating NUL located at string + pmatch[0].rm_eo (there need not actually be a NUL at that location), regardless of the value of nmatch. See below for the definition of pmatch and nmatch. This is an extension, compatible with but not specified by IEEE Std 1003.2 (``POSIX.2''), and should be used with caution in software intended to be portable to other systems. Note that a non-zero rm_so does not imply REG_NOTBOL; REG_STARTEND affects only the location of the string, not how it is matched.

See re_format(7) for a discussion of what is matched in situations where an RE or a portion thereof could match any of several substrings of string.

Normally, regexec() returns 0 for success and the non-zero code REG_NOMATCH for failure. Other non-zero error codes may be returned in exceptional situations; see DIAGNOSTICS.

If REG_NOSUB was specified in the compilation of the RE, or if nmatch is 0, regexec() ignores the pmatch argument (but see below for the case where REG_STARTEND is specified). Otherwise, pmatch points to an array of nmatch structures of type regmatch_t. Such a structure has at least the members rm_so and rm_eo, both of type regoff_t (a signed arithmetic type at least as large as an off_t and a ssize_t), containing respectively the offset of the first character of a substring and the offset of the first character after the end of the substring. Offsets are measured from the beginning of the string argument given to regexec(). An empty substring is denoted by equal offsets, both indicating the character following the empty substring.

The 0th member of the pmatch array is filled in to indicate what substring of string was matched by the entire RE. Remaining members report what substring was matched by parenthesized subexpressions within the RE; member i reports subexpression i, with subexpressions counted (starting at 1) by the order of their opening parentheses in the RE, left to right. Unused entries in the array (corresponding either to subexpressions that did not participate in the match at all, or to subexpressions that do not exist in the RE (that is, i > preg->re_nsub)) have both rm_so and rm_eo set to -1. If a subexpression participated in the match several times, the reported substring is the last one it matched. (Note, as an example in particular, that when the RE `(b*)+' matches `bbb', the parenthesized subexpression matches each of the three `b's and then an infinite number of empty strings following the last `b', so the reported substring is one of the empties.)

If REG_STARTEND is specified, pmatch must point to at least one regmatch_t (even if nmatch is 0 or REG_NOSUB was specified), to hold the input offsets for REG_STARTEND. Use for output is still entirely controlled by nmatch; if nmatch is 0 or REG_NOSUB was specified, the value of pmatch[0] will not be changed by a successful regexec().

The regerror() function maps a non-zero errcode from either regcomp() or regexec() to a human-readable, printable message. If preg is non-NULL, the error code should have arisen from use of the regex_t pointed to by preg, and if the error code came from regcomp(), it should have been the result from the most recent regcomp() using that regex_t. The (regerror() may be able to supply a more detailed message using information from the regex_t.) The regerror() function places the NUL-terminated message into the buffer pointed to by errbuf, limiting the length (including the NUL) to at most errbuf_size bytes. If the whole message will not fit, as much of it as will fit before the terminating NUL is supplied. In any case, the returned value is the size of buffer needed to hold the whole message (including terminating NUL). If errbuf_size is 0, errbuf is ignored but the return value is still correct.

If the errcode given to regerror() is first ORed with REG_ITOA, the ``message'' that results is the printable name of the error code, e.g. ``REG_NOMATCH'', rather than an explanation thereof. If errcode is REG_ATOI, then preg shall be non-NULL and the re_endp member of the structure it points to must point to the printable name of an error code; in this case, the result in errbuf is the decimal digits of the numeric value of the error code (0 if the name is not recognized). REG_ITOA and REG_ATOI are intended primarily as debugging facilities; they are extensions, compatible with but not specified by IEEE Std 1003.2 (``POSIX.2''), and should be used with caution in software intended to be portable to other systems. Be warned also that they are considered experimental and changes are possible.

The regfree() function frees any dynamically-allocated storage associated with the compiled RE pointed to by preg. The remaining regex_t is no longer a valid compiled RE and the effect of supplying it to regexec() or regerror() is undefined.

None of these functions references global variables except for tables of constants; all are safe for use from multiple threads if the arguments are safe.

IMPLEMENTATION CHOICES

There are a number of decisions that IEEE Std 1003.2 (``POSIX.2'') leaves up to the implementor, either by explicitly saying ``undefined'' or by virtue of them being forbidden by the RE grammar. This implementation treats them as follows.

See re_format(7) for a discussion of the definition of case-independent matching.

There is no particular limit on the length of REs, except insofar as memory is limited. Memory usage is approximately linear in RE size, and largely insensitive to RE complexity, except for bounded repetitions. See BUGS for one short RE using them that will run almost any system out of memory.

A backslashed character other than one specifically given a magic meaning by IEEE Std 1003.2 (``POSIX.2'') (such magic meanings occur only in obsolete [``basic''] REs) is taken as an ordinary character.

Any unmatched `[' is a REG_EBRACK error.

Equivalence classes cannot begin or end bracket-expression ranges. The endpoint of one range cannot begin another.

RE_DUP_MAX, the limit on repetition counts in bounded repetitions, is 255.

A repetition operator (`?', `*', `+', or bounds) cannot follow another repetition operator. A repetition operator cannot begin an expression or subexpression or follow `^' or `|'.

`|' cannot appear first or last in a (sub)expression or after another `|', i.e., an operand of `|' cannot be an empty subexpression. An empty parenthesized subexpression, `()', is legal and matches an empty (sub)string. An empty string is not a legal RE.

A `{' followed by a digit is considered the beginning of bounds for a bounded repetition, which must then follow the syntax for bounds. A `{' not followed by a digit is considered an ordinary character.

`^' and `$' beginning and ending subexpressions in obsolete (``basic'') REs are anchors, not ordinary characters.

DIAGNOSTICS

Non-zero error codes from regcomp() and regexec() include the following:

REG_NOMATCH	The regexec() function failed to match
REG_BADPAT	invalid regular expression
REG_ECOLLATE	invalid collating element
REG_ECTYPE	invalid character class
REG_EESCAPE	`\' applied to unescapable character
REG_ESUBREG	invalid backreference number
REG_EBRACK	brackets `[ ]' not balanced
REG_EPAREN	parentheses `( )' not balanced
REG_EBRACE	braces `{ }' not balanced
REG_BADBR	invalid repetition count(s) in `{ }'
REG_ERANGE	invalid character range in `[ ]'
REG_ESPACE	ran out of memory
REG_BADRPT	`?', `*', or `+' operand invalid
REG_EMPTY	empty (sub)expression
REG_ASSERT	cannot happen - you found a bug
REG_INVARG	invalid argument, e.g. negative-length string
REG_ILLSEQ	illegal byte sequence (bad multibyte character)

SEE ALSO

grep(1), re_format(7)

IEEE Std 1003.2 (``POSIX.2''), sections 2.8 (Regular Expression Notation) and B.5 (C Binding for Regular Expression Matching).

HISTORY

Originally written by Henry Spencer. Altered for inclusion in the 4.4BSD distribution.

BUGS

This is an alpha release with known defects. Please report problems.

The back-reference code is subtle and doubts linger about its correctness in complex cases.

The regexec() function performance is poor. This will improve with later releases. The nmatch argument exceeding 0 is expensive; nmatch exceeding 1 is worse. The regexec() function is largely insensitive to RE complexity except that back references are massively expensive. RE length does matter; in particular, there is a strong speed bonus for keeping RE length under about 30 characters, with most special characters counting roughly double.

The regcomp() function implements bounded repetitions by macro expansion, which is costly in time and space if counts are large or bounded repetitions are nested. An RE like, say, `((((a{1,100}){1,100}){1,100}){1,100}){1,100}' will (eventually) run almost any existing machine out of swap space.

There are suspected problems with response to obscure error conditions. Notably, certain kinds of internal overflow, produced only by truly enormous REs or by multiply nested bounded repetitions, are probably not handled well.

Due to a mistake in IEEE Std 1003.2 (``POSIX.2''), things like `a)b' are legal REs because `)' is a special character only in the presence of a previous unmatched `('. This cannot be fixed until the spec is fixed.

The standard's definition of back references is vague. For example, does `a\(\(b\)*\2\)*d' match `abbbd'? Until the standard is clarified, behavior in such cases should not be relied on.

The implementation of word-boundary matching is a bit of a kludge, and bugs may lurk in combinations of word-boundary matching and anchoring.

Word-boundary matching does not work properly in multibyte locales.