NAME

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

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);

DESCRIPTION

These routines implement POSIX 1003.2 regular expressions (‘‘RE’’s); see *regex*(7). *Regcomp* 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_’’.

Regcomp 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*. *Cflags* 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. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems.

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 POSIX 1003.2, 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 *regex*(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 POSIX 1003.2, 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.

Regexec 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 com-

piled 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, with the NUL indicating the end of the line. (That is, any other end-of-line marker is considered to have been removed and replaced by the NUL.) 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 POSIX 1003.2, 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 *regex*(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 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*.

Regerror 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*. (*Regerror* may be able to supply a more detailed message using information from the *regex_t*.) *Regerror* 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 won't 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 POSIX 1003.2, 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.

Regfree 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 1003.2 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 *regex*(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 1003.2 (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.

SEE ALSO

grep(1), regex(7)

POSIX 1003.2, sections 2.8 (Regular Expression Notation) and B.5 (C Binding for Regular Expression Matching).

DIAGNOSTICS

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

REG_NOMATCH	regexec() 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	‘‘can’t happen’’—you found a bug
REG_INVARG	invalid argument, e.g. negative-length string

HISTORY

Written by Henry Spencer, henry@zoo.toronto.edu.

BUGS

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

There is one known functionality bug. The implementation of internationalization is incomplete: the locale is always assumed to be the default one of 1003.2, and only the collating elements etc. of that locale are available.

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

Regexec performance is poor. This will improve with later releases. *Nmatch* exceeding 0 is expensive; *nmatch* exceeding 1 is worse. *Regexec* 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.

Regcomp 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 1003.2, things like ‘a)b’ are legal REs because ‘)’ is a special character only in the presence of a previous unmatched ‘(’. This can’t 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.

NAME

regex − POSIX 1003.2 regular expressions

DESCRIPTION

Regular expressions (‘‘RE’’s), as defined in POSIX 1003.2, come in two forms: modern REs (roughly those of *egrep*; 1003.2 calls these ‘‘extended’’ REs) and obsolete REs (roughly those of *ed*; 1003.2 ‘‘basic’’ REs). Obsolete REs mostly exist for backward compatibility in some old programs; they will be discussed at the end. 1003.2 leaves some aspects of RE syntax and semantics open; ‘†’ marks decisions on these aspects that may not be fully portable to other 1003.2 implementations.

A (modern) RE is one† or more non-empty† *branches*, separated by ‘|’. It matches anything that matches one of the branches.

A branch is one† or more *pieces*, concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an *atom* possibly followed by a single† ‘*’, ‘+’, ‘?’, or *bound*. An atom followed by ‘*’ matches a sequence of 0 or more matches of the atom. An atom followed by ‘+’ matches a sequence of 1 or more matches of the atom. An atom followed by ‘?’ matches a sequence of 0 or 1 matches of the atom.

A *bound* is ‘{’ followed by an unsigned decimal integer, possibly followed by ‘,’ possibly followed by another unsigned decimal integer, always followed by ‘}’. The integers must lie between 0 and RE_DUP_MAX (255†) inclusive, and if there are two of them, the first may not exceed the second. An atom followed by a bound containing one integer *i* and no comma matches a sequence of exactly *i* matches of the atom. An atom followed by a bound containing one integer *i* and a comma matches a sequence of *i* or more matches of the atom. An atom followed by a bound containing two integers *i* and *j* matches a sequence of *i* through *j* (inclusive) matches of the atom.

An atom is a regular expression enclosed in ‘()’ (matching a match for the regular expression), an empty set of ‘()’ (matching the null string)†, a *bracket expression* (see below), ‘.’ (matching any single character), ‘^’ (matching the null string at the beginning of a line), ‘$’ (matching the null string at the end of a line), a ‘\’ followed by one of the characters ‘^.[$()|*+?{\’ (matching that character taken as an ordinary character), a ‘\’ followed by any other character† (matching that character taken as an ordinary character, as if the ‘\’ had not been present†), or a single character with no other significance (matching that character). A ‘{’ followed by a character other than a digit is an ordinary character, not the beginning of a bound†. It is illegal to end an RE with ‘\’.

A *bracket expression* is a list of characters enclosed in ‘[]’. It normally matches any single character from the list (but see below). If the list begins with ‘^’, it matches any single character (but see below) *not* from the rest of the list. If two characters in the list are separated by ‘−’, this is shorthand for the full *range* of characters between those two (inclusive) in the collating sequence, e.g. ‘[0−9]’ in ASCII matches any decimal digit. It is illegal† for two ranges to share an endpoint, e.g. ‘a−c−e’. Ranges are very collating-sequence-dependent, and portable programs should avoid relying on them.

To include a literal ‘]’ in the list, make it the first character (following a possible ‘^’). To include a literal ‘−’, make it the first or last character, or the second endpoint of a range. To use a literal ‘−’ as the first endpoint of a range, enclose it in ‘[.’ and ‘.]’ to make it a collating element (see below). With the exception of these and some combinations using ‘[’ (see next paragraphs), all other special characters, including ‘\’, lose their special significance within a bracket expression.

Within a bracket expression, a collating element (a character, a multi-character sequence that collates as if it were a single character, or a collating-sequence name for either) enclosed in ‘[.’ and ‘.]’ stands for the sequence of characters of that collating element. The sequence is a single element of the bracket expression’s list. A bracket expression containing a multi-character collating element can thus match more than one character, e.g. if the collating sequence includes a ‘ch’ collating element, then the RE ‘[[.ch.]]*c’ matches the first five characters of ‘chchcc’.

Within a bracket expression, a collating element enclosed in ‘[=’ and ‘=]’ is an equivalence class, standing for the sequences of characters of all collating elements equivalent to that one, including itself. (If there are no other equivalent collating elements, the treatment is as if the enclosing delimiters were ‘[.’ and ‘.]’.) For

example, if o and ô are the members of an equivalence class, then '[[=o=]]', '[[=ô=]]', and '[oô]' are all synonymous. An equivalence class may not† be an endpoint of a range.

Within a bracket expression, the name of a *character class* enclosed in '[:' and ':]' stands for the list of all characters belonging to that class. Standard character class names are:

alnum	digit	punct
alpha	graph	space
blank	lower	upper
cntrl	print	xdigit

These stand for the character classes defined in *ctype*(3). A locale may provide others. A character class may not be used as an endpoint of a range.

There are two special cases† of bracket expressions: the bracket expressions '[[:<:]]' and '[[:>:]]' match the null string at the beginning and end of a word respectively. A word is defined as a sequence of word characters which is neither preceded nor followed by word characters. A word character is an *alnum* character (as defined by *ctype*(3)) or an underscore. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems.

In the event that an RE could match more than one substring of a given string, the RE matches the one starting earliest in the string. If the RE could match more than one substring starting at that point, it matches the longest. Subexpressions also match the longest possible substrings, subject to the constraint that the whole match be as long as possible, with subexpressions starting earlier in the RE taking priority over ones starting later. Note that higher-level subexpressions thus take priority over their lower-level component subexpressions.

Match lengths are measured in characters, not collating elements. A null string is considered longer than no match at all. For example, 'bb*' matches the three middle characters of 'abbbc', '(wee|week)(knights|nights)' matches all ten characters of 'weeknights', when '(.*).*' is matched against 'abc' the parenthesized subexpression matches all three characters, and when '(a*)*' is matched against 'bc' both the whole RE and the parenthesized subexpression match the null string.

If case-independent matching is specified, the effect is much as if all case distinctions had vanished from the alphabet. When an alphabetic that exists in multiple cases appears as an ordinary character outside a bracket expression, it is effectively transformed into a bracket expression containing both cases, e.g. 'x' becomes '[xX]'. When it appears inside a bracket expression, all case counterparts of it are added to the bracket expression, so that (e.g.) '[x]' becomes '[xX]' and '[^x]' becomes '[^xX]'.

No particular limit is imposed on the length of REs†. Programs intended to be portable should not employ REs longer than 256 bytes, as an implementation can refuse to accept such REs and remain POSIX-compliant.

Obsolete (''basic'') regular expressions differ in several respects. '|', '+', and '?' are ordinary characters and there is no equivalent for their functionality. The delimiters for bounds are '\{' and '\}', with '{' and '}' by themselves ordinary characters. The parentheses for nested subexpressions are '\(' and '\)', with '(' and ')' by themselves ordinary characters. '^' is an ordinary character except at the beginning of the RE or† the beginning of a parenthesized subexpression, '$' is an ordinary character except at the end of the RE or† the end of a parenthesized subexpression, and '*' is an ordinary character if it appears at the beginning of the RE or the beginning of a parenthesized subexpression (after a possible leading '^'). Finally, there is one new type of atom, a *back reference*: '\' followed by a non-zero decimal digit *d* matches the same sequence of characters matched by the *d*th parenthesized subexpression (numbering subexpressions by the positions of their opening parentheses, left to right), so that (e.g.) '\([bc]\)\1' matches 'bb' or 'cc' but not 'bc'.

SEE ALSO

regex(3)

POSIX 1003.2, section 2.8 (Regular Expression Notation).

HISTORY

Written by Henry Spencer, based on the 1003.2 spec.

BUGS

Having two kinds of REs is a botch.

The current 1003.2 spec says that ‘)’ is an ordinary character in the absence of an unmatched ‘(’; this was an unintentional result of a wording error, and change is likely. Avoid relying on it.

Back references are a dreadful botch, posing major problems for efficient implementations. They are also somewhat vaguely defined (does ‘a\(\(b\)*\2\)*d’ match ‘abbbd’?). Avoid using them.

1003.2’s specification of case-independent matching is vague. The ‘‘one case implies all cases’’ definition given above is current consensus among implementors as to the right interpretation.

The syntax for word boundaries is incredibly ugly.