NAME

PCRE - Perl-compatible regular expressions.

SYNOPSIS OF C++ WRAPPER

#include <pcrecpp.h>

DESCRIPTION

The C++ wrapper for PCRE was provided by Google Inc. Some additional functionality was added by Giuseppe Maxia. This brief man page was constructed from the notes in the *pcrecpp.h* file, which should be consulted for further details. Note that the C++ wrapper supports only the original 8-bit PCRE library. There is no 16-bit or 32-bit support at present.

MATCHING INTERFACE

The "FullMatch" operation checks that supplied text matches a supplied pattern exactly. If pointer arguments are supplied, it copies matched sub-strings that match sub-patterns into them.

```
Example: successful match
  pcrecpp::RE re("h.*o");
  re.FullMatch("hello");

Example: unsuccessful match (requires full match):
  pcrecpp::RE re("e");
  !re.FullMatch("hello");

Example: creating a temporary RE object:
  pcrecpp::RE("h.*o").FullMatch("hello");
```

You can pass in a "const char*" or a "string" for "text". The examples below tend to use a const char*. You can, as in the different examples above, store the RE object explicitly in a variable or use a temporary RE object. The examples below use one mode or the other arbitrarily. Either could correctly be used for any of these examples.

You must supply extra pointer arguments to extract matched subpieces.

```
Example: extracts "ruby" into "s" and 1234 into "i"
  int i;
  string s;
  pcrecpp::RE re("(\\w+):(\\d+)");
  re.FullMatch("ruby:1234", &s, &i);

Example: does not try to extract any extra sub-patterns
  re.FullMatch("ruby:1234", &s);

Example: does not try to extract into NULL
  re.FullMatch("ruby:1234", NULL, &i);

Example: integer overflow causes failure
  !re.FullMatch("ruby:1234567891234", NULL, &i);

Example: fails because there aren't enough sub-patterns:
  !pcrecpp::RE("\\w+:\\d+").FullMatch("ruby:1234", &s);

Example: fails because string cannot be stored in integer
  !pcrecpp::RE("(.*)").FullMatch("ruby", &i);
```

The provided pointer arguments can be pointers to any scalar numeric type, or one of:

```
string       (matched piece is copied to string)
StringPiece  (StringPiece is mutated to point to matched piece)
T            (where "bool T::ParseFrom(const char*, int)" exists)
NULL         (the corresponding matched sub-pattern is not copied)
```

The function returns true iff all of the following conditions are satisfied:

a. "text" matches "pattern" exactly;

b. The number of matched sub-patterns is >= number of supplied pointers;

c. The "i"th argument has a suitable type for holding the string captured as the "i"th sub-pattern. If you pass in void * NULL for the "i"th argument, or a non-void * NULL of the correct type, or pass fewer arguments than the number of sub-patterns, "i"th captured sub-pattern is ignored.

CAVEAT: An optional sub-pattern that does not exist in the matched string is assigned the empty string. Therefore, the following will return false (because the empty string is not a valid number):

```
int number;
pcrecpp::RE::FullMatch("abc", "[a-z]+(\\d+)?", &number);
```

The matching interface supports at most 16 arguments per call. If you need more, consider using the more general interface **pcrecpp::RE::DoMatch**. See **pcrecpp.h** for the signature for **DoMatch**.

NOTE: Do not use **no_arg**, which is used internally to mark the end of a list of optional arguments, as a placeholder for missing arguments, as this can lead to segfaults.

QUOTING METACHARACTERS

You can use the "QuoteMeta" operation to insert backslashes before all potentially meaningful characters in a string. The returned string, used as a regular expression, will exactly match the original string.

Example:

```
string quoted = RE::QuoteMeta(unquoted);
```

Note that it's legal to escape a character even if it has no special meaning in a regular expression -- so this function does that. (This also makes it identical to the perl function of the same name; see "perldoc -f quotemeta".) For example, "1.5-2.0?" becomes "1\.5\-2\.0\?".

PARTIAL MATCHES

You can use the "PartialMatch" operation when you want the pattern to match any substring of the text.

Example: simple search for a string:

```
pcrecpp::RE("ell").PartialMatch("hello");
```

Example: find first number in a string:

```
int number;
pcrecpp::RE re("(\\d+)");
re.PartialMatch("x*100 + 20", &number);
```

```
assert(number == 100);
```

UTF-8 AND THE MATCHING INTERFACE

By default, pattern and text are plain text, one byte per character. The UTF8 flag, passed to the constructor, causes both pattern and string to be treated as UTF-8 text, still a byte stream but potentially multiple bytes per character. In practice, the text is likelier to be UTF-8 than the pattern, but the match returned may depend on the UTF8 flag, so always use it when matching UTF8 text. For example, "." will match one byte normally but with UTF8 set may match up to three bytes of a multi-byte character.

Example:

```
pcrecpp::RE_Options options;
options.set_utf8();
pcrecpp::RE re(utf8_pattern, options);
re.FullMatch(utf8_string);
```

Example: using the convenience function UTF8():

```
pcrecpp::RE re(utf8_pattern, pcrecpp::UTF8());
re.FullMatch(utf8_string);
```

NOTE: The UTF8 flag is ignored if pcre was not configured with the --enable-utf8 flag.

PASSING MODIFIERS TO THE REGULAR EXPRESSION ENGINE

PCRE defines some modifiers to change the behavior of the regular expression engine. The C++ wrapper defines an auxiliary class, RE_Options, as a vehicle to pass such modifiers to a RE class. Currently, the following modifiers are supported:

modifier	description	Perl corresponding
PCRE_CASELESS	case insensitive match	/i
PCRE_MULTILINE	multiple lines match	/m
PCRE_DOTALL	dot matches newlines	/s
PCRE_DOLLAR_ENDONLY	$ matches only at end	N/A
PCRE_EXTRA	strict escape parsing	N/A
PCRE_EXTENDED	ignore white spaces	/x
PCRE_UTF8	handles UTF8 chars	built-in
PCRE_UNGREEDY	reverses * and *?	N/A
PCRE_NO_AUTO_CAPTURE	disables capturing parens	N/A (*)

(*) Both Perl and PCRE allow non capturing parentheses by means of the "?:" modifier within the pattern itself. e.g. (?:ab|cd) does not capture, while (ab|cd) does.

For a full account on how each modifier works, please check the PCRE API reference page.

For each modifier, there are two member functions whose name is made out of the modifier in lowercase, without the "PCRE_" prefix. For instance, PCRE_CASELESS is handled by

```
bool caseless()
```

which returns true if the modifier is set, and

```
RE_Options & set_caseless(bool)
```

which sets or unsets the modifier. Moreover, PCRE_EXTRA_MATCH_LIMIT can be accessed through the **set_match_limit()** and **match_limit()** member functions. Setting *match_limit* to a non-zero value will limit

the execution of pcre to keep it from doing bad things like blowing the stack or taking an eternity to return a result. A value of 5000 is good enough to stop stack blowup in a 2MB thread stack. Setting *match_limit* to zero disables match limiting. Alternatively, you can call **match_limit_recursion()** which uses PCRE_EXTRA_MATCH_LIMIT_RECURSION to limit how much PCRE recurses. **match_limit()** limits the number of matches PCRE does; **match_limit_recursion()** limits the depth of internal recursion, and therefore the amount of stack that is used.

Normally, to pass one or more modifiers to a RE class, you declare a *RE_Options* object, set the appropriate options, and pass this object to a RE constructor. Example:

```
RE_Options opt;
opt.set_caseless(true);
if (RE("HELLO", opt).PartialMatch("hello world")) ...
```

RE_options has two constructors. The default constructor takes no arguments and creates a set of flags that are off by default. The optional parameter *option_flags* is to facilitate transfer of legacy code from C programs. This lets you do

```
RE(pattern,
  RE_Options(PCRE_CASELESS|PCRE_MULTILINE)).PartialMatch(str);
```

However, new code is better off doing

```
RE(pattern,
  RE_Options().set_caseless(true).set_multiline(true))
    .PartialMatch(str);
```

If you are going to pass one of the most used modifiers, there are some convenience functions that return a RE_Options class with the appropriate modifier already set: **CASELESS()**, **UTF8()**, **MULTILINE()**, **DOTALL**(), and **EXTENDED()**.

If you need to set several options at once, and you don't want to go through the pains of declaring a RE_Options object and setting several options, there is a parallel method that give you such ability on the fly. You can concatenate several **set_xxxxx()** member functions, since each of them returns a reference to its class object. For example, to pass PCRE_CASELESS, PCRE_EXTENDED, and PCRE_MULTILINE to a RE with one statement, you may write:

```
RE(" ^ xyz \\s+ .* blah$",
  RE_Options()
    .set_caseless(true)
    .set_extended(true)
    .set_multiline(true)).PartialMatch(sometext);
```

SCANNING TEXT INCREMENTALLY

The "Consume" operation may be useful if you want to repeatedly match regular expressions at the front of a string and skip over them as they match. This requires use of the "StringPiece" type, which represents a sub-range of a real string. Like RE, StringPiece is defined in the pcrecpp namespace.

Example: read lines of the form "var = value" from a string.

```
  string contents = ...;               // Fill string somehow
  pcrecpp::StringPiece input(contents);  // Wrap in a StringPiece

  string var;
  int value;
```

```
pcrecpp::RE re("(\\w+) = (\\d+)\n");
while (re.Consume(&input, &var, &value)) {
  ...;
}
```

Each successful call to "Consume" will set "var/value", and also advance "input" so it points past the matched text.

The "FindAndConsume" operation is similar to "Consume" but does not anchor your match at the beginning of the string. For example, you could extract all words from a string by repeatedly calling

```
pcrecpp::RE("(\\w+)").FindAndConsume(&input, &word)
```

PARSING HEX/OCTAL/C-RADIX NUMBERS

By default, if you pass a pointer to a numeric value, the corresponding text is interpreted as a base-10 number. You can instead wrap the pointer with a call to one of the operators Hex(), Octal(), or CRadix() to interpret the text in another base. The CRadix operator interprets C-style "0" (base-8) and "0x" (base-16) prefixes, but defaults to base-10.

```
Example:
  int a, b, c, d;
  pcrecpp::RE re("(.*) (.*) (.*) (.*)");
  re.FullMatch("100 40 0100 0x40",
               pcrecpp::Octal(&a), pcrecpp::Hex(&b),
               pcrecpp::CRadix(&c), pcrecpp::CRadix(&d));
```

will leave 64 in a, b, c, and d.

REPLACING PARTS OF STRINGS

You can replace the first match of "pattern" in "str" with "rewrite". Within "rewrite", backslash-escaped digits (\1 to \9) can be used to insert text matching corresponding parenthesized group from the pattern. \0 in "rewrite" refers to the entire matching text. For example:

```
string s = "yabba dabba doo";
pcrecpp::RE("b+").Replace("d", &s);
```

will leave "s" containing "yada dabba doo". The result is true if the pattern matches and a replacement occurs, false otherwise.

GlobalReplace is like **Replace** except that it replaces all occurrences of the pattern in the string with the rewrite. Replacements are not subject to re-matching. For example:

```
string s = "yabba dabba doo";
pcrecpp::RE("b+").GlobalReplace("d", &s);
```

will leave "s" containing "yada dada doo". It returns the number of replacements made.

Extract is like **Replace**, except that if the pattern matches, "rewrite" is copied into "out" (an additional argument) with substitutions. The non-matching portions of "text" are ignored. Returns true iff a match occurred and the extraction happened successfully; if no match occurs, the string is left unaffected.

AUTHOR

The C++ wrapper was contributed by Google Inc.

REVISION

Last updated: 08 January 2012