NAME

// - A demonstration C program for PCRE2 - //

```
/************************************************
*          PCRE2 DEMONSTRATION PROGRAM         *
************************************************/
```

/* This is a demonstration program to illustrate a straightforward way of
using the PCRE2 regular expression library from a C program. See the
pcre2sample documentation for a short discussion ("man pcre2sample" if you have
the PCRE2 man pages installed). PCRE2 is a revised API for the library, and is
incompatible with the original PCRE API.

There are actually three libraries, each supporting a different code unit
width. This demonstration program uses the 8-bit library. The default is to
process each code unit as a separate character, but if the pattern begins with
"(*UTF)", both it and the subject are treated as UTF-8 strings, where
characters may occupy multiple code units.

In Unix-like environments, if PCRE2 is installed in your standard system
libraries, you should be able to compile this program using this command:

```
cc -Wall pcre2demo.c -lpcre2-8 -o pcre2demo
```

If PCRE2 is not installed in a standard place, it is likely to be installed
with support for the pkg-config mechanism. If you have pkg-config, you can
compile this program using this command:

```
cc -Wall pcre2demo.c ‘pkg-config --cflags --libs libpcre2-8‘ -o pcre2demo
```

If you do not have pkg-config, you may have to use something like this:

```
cc -Wall pcre2demo.c -I/usr/local/include -L/usr/local/lib \
  -R/usr/local/lib -lpcre2-8 -o pcre2demo
```

Replace "/usr/local/include" and "/usr/local/lib" with wherever the include and
library files for PCRE2 are installed on your system. Only some operating
systems (Solaris is one) use the -R option.

Building under Windows:

```
If you want to statically link this program against a non-dll .a file, you must
define PCRE2_STATIC before including pcre2.h, so in this environment, uncomment
the following line. */

/* #define PCRE2_STATIC */

/* The PCRE2_CODE_UNIT_WIDTH macro must be defined before including pcre2.h.
For a program that uses only one code unit width, setting it to 8, 16, or 32
makes it possible to use generic function names such as pcre2_compile(). Note
that just changing 8 to 16 (for example) is not sufficient to convert this
program to process 16-bit characters. Even in a fully 16-bit environment, where
string-handling functions such as strcmp() and printf() work with 16-bit
characters, the code for handling the table of named substrings will still need
to be modified. */

#define PCRE2_CODE_UNIT_WIDTH 8

#include <stdio.h>
#include <string.h>
#include <pcre2.h>


/*************************************************************************
* Here is the program. The API includes the concept of "contexts" for     *
* setting up unusual interface requirements for compiling and matching,   *
* such as custom memory managers and non-standard newline definitions.    *
* This program does not do any of this, so it makes no use of contexts,   *
* always passing NULL where a context could be given.                     *
*************************************************************************/

int main(int argc, char **argv)
{
pcre2_code *re;
PCRE2_SPTR pattern;     /* PCRE2_SPTR is a pointer to unsigned code units of */
PCRE2_SPTR subject;     /* the appropriate width (in this case, 8 bits). */
PCRE2_SPTR name_table;

int crlf_is_newline;
int errornumber;
int find_all;
```

```
int i;
int rc;
int utf8;

uint32_t option_bits;
uint32_t namecount;
uint32_t name_entry_size;
uint32_t newline;

PCRE2_SIZE erroroffset;
PCRE2_SIZE *ovector;
PCRE2_SIZE subject_length;

pcre2_match_data *match_data;


/*************************************************************************
* First, sort out the command line. There is only one possible option at  *
* the moment, "-g" to request repeated matching to find all occurrences,  *
* like Perl's /g option. We set the variable find_all to a non-zero value *
* if the -g option is present.                                            *
*************************************************************************/

find_all = 0;
for (i = 1; i < argc; i++)
  {
  if (strcmp(argv[i], "-g") == 0) find_all = 1;
  else if (argv[i][0] == '-')
    {
    printf("Unrecognised option %s\n", argv[i]);
    return 1;
    }
  else break;
  }

/* After the options, we require exactly two arguments, which are the pattern,
and the subject string. */

if (argc - i != 2)
  {
```

```
  printf("Exactly two arguments required: a regex and a subject string\n");
  return 1;
  }

/* Pattern and subject are char arguments, so they can be straightforwardly
cast to PCRE2_SPTR because we are working in 8-bit code units. The subject
length is cast to PCRE2_SIZE for completeness, though PCRE2_SIZE is in fact
defined to be size_t. */

pattern = (PCRE2_SPTR)argv[i];
subject = (PCRE2_SPTR)argv[i+1];
subject_length = (PCRE2_SIZE)strlen((char *)subject);


/*************************************************************************
* Now we are going to compile the regular expression pattern, and handle *
* any errors that are detected.                                          *
*************************************************************************/

re = pcre2_compile(
  pattern,              /* the pattern */
  PCRE2_ZERO_TERMINATED, /* indicates pattern is zero-terminated */
  0,                    /* default options */
  &errornumber,         /* for error number */
  &erroroffset,         /* for error offset */
  NULL);                /* use default compile context */

/* Compilation failed: print the error message and exit. */

if (re == NULL)
  {
  PCRE2_UCHAR buffer[256];
  pcre2_get_error_message(errornumber, buffer, sizeof(buffer));
  printf("PCRE2 compilation failed at offset %d: %s\n", (int)erroroffset,
    buffer);
  return 1;
  }


/*************************************************************************
```

```
* If the compilation succeeded, we call PCRE2 again, in order to do a    *
* pattern match against the subject string. This does just ONE match. If *
* further matching is needed, it will be done below. Before running the  *
* match we must set up a match_data block for holding the result. Using  *
* pcre2_match_data_create_from_pattern() ensures that the block is       *
* exactly the right size for the number of capturing parentheses in the  *
* pattern. If you need to know the actual size of a match_data block as  *
* a number of bytes, you can find it like this:                          *
*                                                                        *
* PCRE2_SIZE match_data_size = pcre2_get_match_data_size(match_data);    *
**************************************************************************/

match_data = pcre2_match_data_create_from_pattern(re, NULL);

/* Now run the match. */

rc = pcre2_match(
  re,                /* the compiled pattern */
  subject,             /* the subject string */
  subject_length,       /* the length of the subject */
  0,                 /* start at offset 0 in the subject */
  0,                 /* default options */
  match_data,          /* block for storing the result */
  NULL);              /* use default match context */

/* Matching failed: handle error cases */

if (rc < 0)
  {
  switch(rc)
    {
    case PCRE2_ERROR_NOMATCH: printf("No match\n"); break;
    /*
    Handle other special cases if you like
    */
    default: printf("Matching error %d\n", rc); break;
    }
  pcre2_match_data_free(match_data);   /* Release memory used for the match */
  pcre2_code_free(re);                 /*   data and the compiled pattern. */
  return 1;
```

```
  }

/* Match succeeded. Get a pointer to the output vector, where string offsets
are stored. */

ovector = pcre2_get_ovector_pointer(match_data);
printf("Match succeeded at offset %d\n", (int)ovector[0]);


/*************************************************************************
* We have found the first match within the subject string. If the output *
* vector wasn't big enough, say so. Then output any substrings that were *
* captured.                                                              *
*************************************************************************/

/* The output vector wasn't big enough. This should not happen, because we used
pcre2_match_data_create_from_pattern() above. */

if (rc == 0)
  printf("ovector was not big enough for all the captured substrings\n");

/* Since release 10.38 PCRE2 has locked out the use of \K in lookaround
assertions. However, there is an option to re-enable the old behaviour. If that
is set, it is possible to run patterns such as /(?=.\K)/ that use \K in an
assertion to set the start of a match later than its end. In this demonstration
program, we show how to detect this case, but it shouldn't arise because the
option is never set. */

if (ovector[0] > ovector[1])
  {
  printf("\\K was used in an assertion to set the match start after its end.\n"
    "From end to start the match was: %.*s\n", (int)(ovector[0] - ovector[1]),
      (char *)(subject + ovector[1]));
  printf("Run abandoned\n");
  pcre2_match_data_free(match_data);
  pcre2_code_free(re);
  return 1;
  }

/* Show substrings stored in the output vector by number. Obviously, in a real
```

```
application you might want to do things other than print them. */

for (i = 0; i < rc; i++)
  {
  PCRE2_SPTR substring_start = subject + ovector[2*i];
  PCRE2_SIZE substring_length = ovector[2*i+1] - ovector[2*i];
  printf("%2d: %.*s\n", i, (int)substring_length, (char *)substring_start);
  }


/**************************************************************************
* That concludes the basic part of this demonstration program. We have    *
* compiled a pattern, and performed a single match. The code that follows *
* shows first how to access named substrings, and then how to code for    *
* repeated matches on the same subject.                                   *
**************************************************************************/

/* See if there are any named substrings, and if so, show them by name. First
we have to extract the count of named parentheses from the pattern. */

(void)pcre2_pattern_info(
  re,               /* the compiled pattern */
  PCRE2_INFO_NAMECOUNT, /* get the number of named substrings */
  &namecount);      /* where to put the answer */

if (namecount == 0) printf("No named substrings\n"); else
  {
  PCRE2_SPTR tabptr;
  printf("Named substrings\n");

  /* Before we can access the substrings, we must extract the table for
  translating names to numbers, and the size of each entry in the table. */

  (void)pcre2_pattern_info(
    re,                 /* the compiled pattern */
    PCRE2_INFO_NAMETABLE,     /* address of the table */
    &name_table);       /* where to put the answer */

  (void)pcre2_pattern_info(
    re,                 /* the compiled pattern */
```

```
  PCRE2_INFO_NAMEENTRYSIZE, /* size of each entry in the table */
  &name_entry_size);       /* where to put the answer */

/* Now we can scan the table and, for each entry, print the number, the name,
and the substring itself. In the 8-bit library the number is held in two
bytes, most significant first. */

tabptr = name_table;
for (i = 0; i < namecount; i++)
  {
  int n = (tabptr[0] << 8) | tabptr[1];
  printf("(%d) %*s: %.*s\n", n, name_entry_size - 3, tabptr + 2,
    (int)(ovector[2*n+1] - ovector[2*n]), subject + ovector[2*n]);
  tabptr += name_entry_size;
  }
}


/*************************************************************************
* If the "-g" option was given on the command line, we want to continue  *
* to search for additional matches in the subject string, in a similar   *
* way to the /g option in Perl. This turns out to be trickier than you   *
* might think because of the possibility of matching an empty string.    *
* What happens is as follows:                                            *
*                                                                        *
* If the previous match was NOT for an empty string, we can just start   *
* the next match at the end of the previous one.                         *
*                                                                        *
* If the previous match WAS for an empty string, we can't do that, as it *
* would lead to an infinite loop. Instead, a call of pcre2_match() is    *
* made with the PCRE2_NOTEMPTY_ATSTART and PCRE2_ANCHORED flags set. The *
* first of these tells PCRE2 that an empty string at the start of the    *
* subject is not a valid match; other possibilities must be tried. The   *
* second flag restricts PCRE2 to one match attempt at the initial string *
* position. If this match succeeds, an alternative to the empty string   *
* match has been found, and we can print it and proceed round the loop,  *
* advancing by the length of whatever was found. If this match does not  *
* succeed, we still stay in the loop, advancing by just one character.   *
* In UTF-8 mode, which can be set by (*UTF) in the pattern, this may be  *
* more than one byte.                                                    *
```

```
*                                                    *
* However, there is a complication concerned with newlines. When the     *
* newline convention is such that CRLF is a valid newline, we must       *
* advance by two characters rather than one. The newline convention can  *
* be set in the regex by (*CR), etc.; if not, we must find the default.  *
*************************************************************************/

if (!find_all)     /* Check for -g */
  {
  pcre2_match_data_free(match_data);  /* Release the memory that was used */
  pcre2_code_free(re);                /* for the match data and the pattern. */
  return 0;                           /* Exit the program. */
  }

/* Before running the loop, check for UTF-8 and whether CRLF is a valid newline
sequence. First, find the options with which the regex was compiled and extract
the UTF state. */

(void)pcre2_pattern_info(re, PCRE2_INFO_ALLOPTIONS, &option_bits);
utf8 = (option_bits & PCRE2_UTF) != 0;

/* Now find the newline convention and see whether CRLF is a valid newline
sequence. */

(void)pcre2_pattern_info(re, PCRE2_INFO_NEWLINE, &newline);
crlf_is_newline = newline == PCRE2_NEWLINE_ANY ||
                  newline == PCRE2_NEWLINE_CRLF ||
                  newline == PCRE2_NEWLINE_ANYCRLF;

/* Loop for second and subsequent matches */

for (;;)
  {
  uint32_t options = 0;                   /* Normally no options */
  PCRE2_SIZE start_offset = ovector[1];   /* Start at end of previous match */

  /* If the previous match was for an empty string, we are finished if we are
  at the end of the subject. Otherwise, arrange to run another match at the
  same point to see if a non-empty match can be found. */
```

```
if (ovector[0] == ovector[1])
  {
  if (ovector[0] == subject_length) break;
  options = PCRE2_NOTEMPTY_ATSTART | PCRE2_ANCHORED;
  }

/* If the previous match was not an empty string, there is one tricky case to
consider. If a pattern contains \K within a lookbehind assertion at the
start, the end of the matched string can be at the offset where the match
started. Without special action, this leads to a loop that keeps on matching
the same substring. We must detect this case and arrange to move the start on
by one character. The pcre2_get_startchar() function returns the starting
offset that was passed to pcre2_match(). */

else
  {
  PCRE2_SIZE startchar = pcre2_get_startchar(match_data);
  if (start_offset <= startchar)
    {
    if (startchar >= subject_length) break;   /* Reached end of subject.   */
    start_offset = startchar + 1;             /* Advance by one character. */
    if (utf8)                                 /* If UTF-8, it may be more  */
      {                                       /*   than one code unit.     */
      for (; start_offset < subject_length; start_offset++)
        if ((subject[start_offset] & 0xc0) != 0x80) break;
      }
    }
  }

/* Run the next matching operation */

rc = pcre2_match(
  re,                   /* the compiled pattern */
  subject,              /* the subject string */
  subject_length,       /* the length of the subject */
  start_offset,         /* starting offset in the subject */
  options,              /* options */
  match_data,           /* block for storing the result */
  NULL);                /* use default match context */
```

```
/* This time, a result of NOMATCH isn't an error. If the value in "options"
is zero, it just means we have found all possible matches, so the loop ends.
Otherwise, it means we have failed to find a non-empty-string match at a
point where there was a previous empty-string match. In this case, we do what
Perl does: advance the matching position by one character, and continue. We
do this by setting the "end of previous match" offset, because that is picked
up at the top of the loop as the point at which to start again.

There are two complications: (a) When CRLF is a valid newline sequence, and
the current position is just before it, advance by an extra byte. (b)
Otherwise we must ensure that we skip an entire UTF character if we are in
UTF mode. */

if (rc == PCRE2_ERROR_NOMATCH)
  {
  if (options == 0) break;                    /* All matches found */
  ovector[1] = start_offset + 1;              /* Advance one code unit */
  if (crlf_is_newline &&                      /* If CRLF is a newline & */
      start_offset < subject_length - 1 &&    /* we are at CRLF, */
      subject[start_offset] == '\r' &&
      subject[start_offset + 1] == '\n')
    ovector[1] += 1;                          /* Advance by one more. */
  else if (utf8)                              /* Otherwise, ensure we */
    {                                         /* advance a whole UTF-8 */
    while (ovector[1] < subject_length)       /* character. */
      {
      if ((subject[ovector[1]] & 0xc0) != 0x80) break;
      ovector[1] += 1;
      }
    }
  continue;    /* Go round the loop again */
  }

/* Other matching errors are not recoverable. */

if (rc < 0)
  {
  printf("Matching error %d\n", rc);
  pcre2_match_data_free(match_data);
  pcre2_code_free(re);
```

```
  return 1;
  }

/* Match succeeded */

printf("\nMatch succeeded again at offset %d\n", (int)ovector[0]);

/* The match succeeded, but the output vector wasn't big enough. This
should not happen. */

if (rc == 0)
  printf("ovector was not big enough for all the captured substrings\n");

/* We must guard against patterns such as /(?=.\K)/ that use \K in an
assertion to set the start of a match later than its end. In this
demonstration program, we just detect this case and give up. */

if (ovector[0] > ovector[1])
  {
  printf("\\K was used in an assertion to set the match start after its end.\n"
    "From end to start the match was: %.*s\n", (int)(ovector[0] - ovector[1]),
      (char *)(subject + ovector[1]));
  printf("Run abandoned\n");
  pcre2_match_data_free(match_data);
  pcre2_code_free(re);
  return 1;
  }

/* As before, show substrings stored in the output vector by number, and then
also any named substrings. */

for (i = 0; i < rc; i++)
  {
  PCRE2_SPTR substring_start = subject + ovector[2*i];
  size_t substring_length = ovector[2*i+1] - ovector[2*i];
  printf("%2d: %.*s\n", i, (int)substring_length, (char *)substring_start);
  }

if (namecount == 0) printf("No named substrings\n"); else
  {
```

```
    PCRE2_SPTR tabptr = name_table;
    printf("Named substrings\n");
    for (i = 0; i < namecount; i++)
      {
      int n = (tabptr[0] << 8) | tabptr[1];
      printf("(%d) %*s: %.*s\n", n, name_entry_size - 3, tabptr + 2,
        (int)(ovector[2*n+1] - ovector[2*n]), subject + ovector[2*n]);
      tabptr += name_entry_size;
      }
    }
  }      /* End of loop to find second and subsequent matches */

printf("\n");
pcre2_match_data_free(match_data);
pcre2_code_free(re);
return 0;
}

/* End of pcre2demo.c */
```