```
/*************************************************
*           PCRE DEMONSTRATION PROGRAM           *
*************************************************/

/* This is a demonstration program to illustrate the most straightforward ways
of calling the PCRE regular expression library from a C program. See the
pcresample documentation for a short discussion ("man pcresample" if you have
the PCRE man pages installed).

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

gcc -Wall pcredemo.c -lpcre -o pcredemo

If PCRE 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:

gcc -Wall pcredemo.c ‘pkg-config --cflags --libs libpcre‘ -o pcredemo

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

gcc -Wall pcredemo.c -I/usr/local/include -L/usr/local/lib \
  -R/usr/local/lib -lpcre -o pcredemo

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

Building under Windows:

If you want to statically link this program against a non-dll .a file, you must
define PCRE_STATIC before including pcre.h, otherwise the pcre_malloc() and
pcre_free() exported functions will be declared __declspec(dllimport), with
unwanted results. So in this environment, uncomment the following line. */

/* #define PCRE_STATIC */

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

```
#include <pcre.h>

#define OVECCOUNT 30    /* should be a multiple of 3 */


int main(int argc, char **argv)
{
pcre *re;
const char *error;
char *pattern;
char *subject;
unsigned char *name_table;
unsigned int option_bits;
int erroffset;
int find_all;
int crlf_is_newline;
int namecount;
int name_entry_size;
int ovector[OVECCOUNT];
int subject_length;
int rc, i;
int utf8;


/**************************************************************************
* 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. Apart from that, there must be exactly two *
* arguments.                                                              *
**************************************************************************/

find_all = 0;
for (i = 1; i < argc; i++)
  {
  if (strcmp(argv[i], "-g") == 0) find_all = 1;
    else break;
  }

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

if (argc - i != 2)
  {
  printf("Two arguments required: a regex and a subject string\n");
  return 1;
  }

pattern = argv[i];
subject = argv[i+1];
subject_length = (int)strlen(subject);


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

re = pcre_compile(
  pattern,              /* the pattern */
  0,                    /* default options */
  &error,               /* for error message */
  &erroffset,           /* for error offset */
  NULL);                /* use default character tables */

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

if (re == NULL)
  {
  printf("PCRE compilation failed at offset %d: %s\n", erroffset, error);
  return 1;
  }


/*************************************************************************
* If the compilation succeeded, we call PCRE 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.                     *
*************************************************************************/
```

```
rc = pcre_exec(
  re,                   /* the compiled pattern */
  NULL,                 /* no extra data - we didn’t study the pattern */
  subject,              /* the subject string */
  subject_length,       /* the length of the subject */
  0,                    /* start at offset 0 in the subject */
  0,                    /* default options */
  ovector,              /* output vector for substring information */
  OVECCOUNT);           /* number of elements in the output vector */

/* Matching failed: handle error cases */

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

/* Match succeeded */

printf("\nMatch succeeded at offset %d\n", 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 */

if (rc == 0)
```

```
  {
  rc = OVECCOUNT/3;
  printf("ovector only has room for %d captured substrings\n", rc - 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++)
  {
  char *substring_start = subject + ovector[2*i];
  int substring_length = ovector[2*i+1] - ovector[2*i];
  printf("%2d: %.*s\n", i, substring_length, 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)pcre_fullinfo(
  re,                   /* the compiled pattern */
  NULL,                 /* no extra data - we didn't study the pattern */
  PCRE_INFO_NAMECOUNT,  /* number of named substrings */
  &namecount);          /* where to put the answer */

if (namecount <= 0) printf("No named substrings\n"); else
  {
  unsigned char *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)pcre_fullinfo(
    re,                       /* the compiled pattern */
    NULL,                     /* no extra data - we didn’t study the pattern */
    PCRE_INFO_NAMETABLE,      /* address of the table */
    &name_table);             /* where to put the answer */

  (void)pcre_fullinfo(
    re,                       /* the compiled pattern */
    NULL,                     /* no extra data - we didn’t study the pattern */
    PCRE_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. */

  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,
      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 special call of pcre_exec() *
* is made with the PCRE_NOTEMPTY_ATSTART and PCRE_ANCHORED flags set.    *
* The first of these tells PCRE that an empty string at the start of the *
* subject is not a valid match; other possibilities must be tried. The   *
* second flag restricts PCRE 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 (*UTF8) 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 */
  {
  pcre_free(re);   /* Release the memory used for the compiled pattern */
  return 0;        /* Finish unless -g was given */
  }

/* 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; extract
the UTF-8 state, and mask off all but the newline options. */

(void)pcre_fullinfo(re, NULL, PCRE_INFO_OPTIONS, &option_bits);
utf8 = option_bits & PCRE_UTF8;
option_bits &= PCRE_NEWLINE_CR|PCRE_NEWLINE_LF|PCRE_NEWLINE_CRLF|
               PCRE_NEWLINE_ANY|PCRE_NEWLINE_ANYCRLF;

/* If no newline options were set, find the default newline convention from the
build configuration. */

if (option_bits == 0)
  {
  int d;
  (void)pcre_config(PCRE_CONFIG_NEWLINE, &d);
  /* Note that these values are always the ASCII ones, even in
  EBCDIC environments. CR = 13, NL = 10. */
  option_bits = (d == 13)? PCRE_NEWLINE_CR :
        (d == 10)? PCRE_NEWLINE_LF :
        (d == (13<<8 | 10))? PCRE_NEWLINE_CRLF :
        (d == -2)? PCRE_NEWLINE_ANYCRLF :
        (d == -1)? PCRE_NEWLINE_ANY : 0;
  }

/* See if CRLF is a valid newline sequence. */

crlf_is_newline =
     option_bits == PCRE_NEWLINE_ANY ||
     option_bits == PCRE_NEWLINE_CRLF ||
     option_bits == PCRE_NEWLINE_ANYCRLF;

/* Loop for second and subsequent matches */

for (;;)
  {
  int options = 0;                 /* Normally no options */
  int 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 = PCRE_NOTEMPTY_ATSTART | PCRE_ANCHORED;
    }

  /* Run the next matching operation */

  rc = pcre_exec(
    re,                   /* the compiled pattern */
    NULL,                 /* no extra data - we didn't study the pattern */
    subject,              /* the subject string */
    subject_length,       /* the length of the subject */
    start_offset,         /* starting offset in the subject */
    options,              /* options */
    ovector,              /* output vector for substring information */
    OVECCOUNT);           /* number of elements in the output vector */

  /* 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-8 character if we are in
  UTF-8 mode. */

  if (rc == PCRE_ERROR_NOMATCH)
    {
    if (options == 0) break;                    /* All matches found */
    ovector[1] = start_offset + 1;              /* Advance one byte */
    if (crlf_is_newline &&                      /* If CRLF is 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);
    pcre_free(re);    /* Release memory used for the compiled pattern */
    return 1;
    }

  /* Match succeeded */

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

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

  if (rc == 0)
    {
    rc = OVECCOUNT/3;
    printf("ovector only has room for %d captured substrings\n", rc - 1);
    }

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

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

  if (namecount <= 0) printf("No named substrings\n"); else
    {
    unsigned char *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,
        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");
pcre_free(re);       /* Release memory used for the compiled pattern */
return 0;
}

/* End of pcredemo.c */
```