Les expressions régulières en C

par Nicolas Joseph

Date de publication : 14 Septembre 2005

Dernière mise à jour : 28 Novemebre 2005

Ce tutoriel aborde l'utilisation des expressions régulières en C

Historique
    28 Novembre 2005
I - Introduction
II - Les fonctions de la bibliothèque regex
    II-A - regex_t
    II-B - regcomp
    II-C - regexec
    II-D - regerror
    II-E - regfree
III - Les expressions régulières
IV - Utilisation
    IV-A - Savoir si une chaîne correspond à un motif
    IV-B - Extraire une sous chaîne correspondant à un motif
V - Conclusion
VI - Autres documentations
VII - Remerciements

Historique

28 Novembre 2005

Rectification d'une fuites mémoire dans le second exemple

Ajout du caractère de fin de ligne après le *strncpy* dans le second exemple

I - Introduction

Les expressions régulières Le terme *expression régulière* est la traduction douteuse de l'anglais *regular expression* encore appelée *expression rationnelle* abrégé en *regex* ou *regexp*. Tout au long de ce tutoriel, je me contenterai du terme le plus couramment utilisé : *expression régulière*. sont des chaînes de caractères qui se contentent de décrire un motif. Elles ont la réputation d'avoir une syntaxe difficile à apprendre et à relire, ce qui, j'espère vous en convaincre, est faux.

Ce tutoriel utilise la bibliothèque regex qui fait partie du projet GNU. Cette bibliothèque nécessite un système compatible POSIX.2 (la compilation s'est déroulée sans problème avec cygwin sous Windows XP).

II - Les fonctions de la bibliothèque regex

II-A - regex_t

Le type *regex_t* est une structure de données destinée à stocker une expression régulière sous forme compilée. Voici la définition de la structure telle qu'elle est déclarée dans *regex.h* :

```
  struct re_pattern_buffer
  {
  /* [[[begin pattern_buffer]]] */
          /* Space that holds the compiled pattern.  It is declared as
             `unsigned char *' because its elements are
             sometimes used as array indexes.  */
    unsigned char *buffer;

          /* Number of bytes to which `buffer' points.  */
    unsigned long allocated;

          /* Number of bytes actually used in `buffer'.  */
    unsigned long used;

          /* Syntax setting with which the pattern was compiled.  */
    reg_syntax_t syntax;

          /* Pointer to a fastmap, if any, otherwise zero.  re_search uses
             the fastmap, if there is one, to skip over impossible
             starting points for matches.  */
    char *fastmap;

          /* Either a translate table to apply to all characters before
             comparing them, or zero for no translation.  The translation
             is applied to a pattern when it is compiled and to a string
             when it is matched.  */
    char *translate;

          /* Number of subexpressions found by the compiler.  */
    size_t re_nsub;

          /* Zero if this pattern cannot match the empty string, one else.
             Well, in truth it's used only in `re_search_2', to see
             whether or not we should use the fastmap, so we don't set
             this absolutely perfectly; see `re_compile_fastmap' (the
             `duplicate' case).  */
    unsigned can_be_null : 1;

          /* If REGS_UNALLOCATED, allocate space in the `regs' structure
               for `max (RE_NREGS, re_nsub + 1)' groups.
             If REGS_REALLOCATE, reallocate space if necessary.
             If REGS_FIXED, use what's there.  */
  #define REGS_UNALLOCATED 0
  #define REGS_REALLOCATE 1
  #define REGS_FIXED 2
    unsigned regs_allocated : 2;

          /* Set to zero when `regex_compile' compiles a pattern; set to one
             by `re_compile_fastmap' if it updates the fastmap.  */
    unsigned fastmap_accurate : 1;

          /* If set, `re_match_2' does not return information about
             subexpressions.  */
    unsigned no_sub : 1;

          /* If set, a beginning-of-line anchor doesn't match at the
             beginning of the string.  */
    unsigned not_bol : 1;

          /* Similarly for an end-of-line anchor.  */
    unsigned not_eol : 1;

          /* If true, an anchor at a newline matches.  */
    unsigned newline_anchor : 1;

  /* [[[end pattern_buffer]]] */
```

```
};
typedef struct re_pattern_buffer regex_t;
```

Dans le cadre de ce tutoriel, seul le champs *re_nsub* va nous intéresser.

II-B - regcomp

Prototype

```
int regcomp (regex_t *preg, const char *regex, int cflags);
```

Cette fonction a pour but de compiler l'expression régulière *regex* fournie sous forme de chaîne de caractères pour la transformer en structure de type *regex_t* dont l'adresse est passée en premier argument. Il est possible de modifier le comportement de cette fonction par l'intermédiaire de *cflags* qui peut être un OU binaire de l'une des constantes suivantes :

- REG_EXTENDED : permet d'utiliser le modèle d'expression régulière étendu plutôt que le mode basique qui est l'option par défaut
- REG_ICASE : permet d'ignorer la casse (minuscules/majuscules)
- REG_NOSUB : compile pour vérifier uniquement la concordance (vrai ou faux)
- REG_NEWLINE : Par défaut, le caractère de fin de ligne est un caractère normal. Avec cette option, les expressions '[^', '.' et '$' (nous reviendrons plus tard sur la signification de ces expressions) incluent le caractère de fin de ligne implicitement. L'ancre '^' reconnaît une chaîne vide après un caractère de fin de ligne.

En cas de succès, la fonction retourne 0. En cas d'échec de la compilation, *regcomp* retourne un code d'erreur non nul.

II-C - regexec

Prototype

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

Une fois notre expression compilée, on va pouvoir la comparer à la chaîne de caractères *string* à l'aide de cette fonction. Le résultat de la comparaison sera stocké dans le tableau *pmatch* alloué à *nmatch* éléments par nos soins. Il est possible de modifier le comportement de *regexec* grâce à *eflags* :

- REG_NOTBOL : le premier caractère de la chaîne échoue toujours. Ceci n'a pas d'effet s'il s'agit du caractère de fin de ligne et que l'option REG_NEWLINE a été utilisée pour compiler l'expression régulière
- REG_NOTEOL : idem sauf qu'il s'agit du dernier caractère. La remarque à propos de REG_NEWLINE est également valable

regexec retourne 0 en cas de succès ou REG_NOMATCH en cas d'échec.

II-D - regerror

Prototype

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

Cette fonction permet d'obtenir la description correspondante au code d'erreur *errcode* retourné par la fonction

regcomp lors de la création de *preg*. Le message d'erreur d'au plus *errbuf_size* - 1 caractères est placé dans le tampon *errbuf*. *regerror* retourne la taille du message d'erreur.

II-E - regfree

Prototype

```
void regfree (regex_t *preg);
```

Libère la mémoire allouée lors de la compilation.

III - Les expressions régulières

La syntaxe des expressions régulières est celle décrite par le programme GNU grep.

Dans l'expression, un caractère vaut lui même (mis à part quelques métacaractères qui devront être échappés) :

```
a
```

Reconnaîtra toutes les chaînes contenant la lettre *a*. Il est bien sur possible de tester la présence de plusieurs caractères placés les uns à la suite des autres :

```
egu
```

Reconnaîtra, par exemple, les chaînes "Legume" et "Expression reguliere". Pour donner le choix entre plusieurs lettres, il suffit de mettre le groupe entre crochets :

```
[mt]oi
```

Toutes chaînes contenant la lettre 'm' **ou** 't' suivie des deux lettres 'o' et 'i' seront valides.

Si l'on souhaite reconnaître l'une des lettres de l'alphabet en minuscule, on pourrait écrire :

```
[abcdefghijklmnopqrstuvwxyz]
```

Mais cela peut vite s'avérer lourd dans une expression qui voudrait tester la validité d'une adresse mail. Il est donc possible de faire des regroupements en n'indiquant uniquement les bornes de l'ensemble séparé par un tiret :

```
[a-z]
```

Revient au même. Il existe un certains nombre d'ensembles prédéfinies dont voici la liste :

Ensemble prédéfini	Correspondance
[:digit:]	0-9
[:alpha:]	A-Za-z
[:alnum:]	0-9A-Za-z
[:cntrl:]	Les caractères de contrôles (code ASCII 0177 et inférieur à 040)
[:print:]	Les caractères imprimables
[:graph:]	Idem [:print:] sans l'espace
[:lower:]	a-z
[:upper:]	A-Z
[:punct:]	Ni [:cntrl:] ni [:alnum:]
[:space:]	\n\t\r\f
[:xdigit:]	0-9a-fA-F nombres hexadécimaux

Ces définitions concordent avec celles que l'on trouve dans le fichiers d'en tête *ctype.h*. Le point '.' permet de reconnaître n'importe quel caractère. Il est aussi possible de préciser le nombre de répétitions que l'on souhaite pour un élément :

Opérateurs	Signification
?	L'élément est répété, au plus une fois
*	L'élément est présent 0 ou plus de fois
+	L'élément est présent au moins une fois
{n}	L'élément est présent exactement *n* fois
{n,}	L'élément est présent au moins *n* fois
{n,m}	L'élément est présent entre *n* et *m* fois

Un élément est un groupe délimité par des crochets qui sont optionnels si le groupe ne comporte qu'un élément. Voici un exemple pour reconnaître si une chaîne contient trois 'a' consécutifs :

```
[a]{3}
```

L'opposé d'une expression est obtenue en la faisant précéder par le caractère '^'. Si l'on souhaite donné le choix entre deux expressions, il suffit de les séparer par le caractère '|'.

Ce même caractère peut être placé au début de l'expression régulière pour préciser que la chaîne à analysée doit commencer par l'élément suivant :

```
^[A-Z]
```

Précise que la chaîne doit commencer par une lettre majuscule. Le caractère '$' à le même rôle mais cette fois en fin de chaîne.

Pour finir voici la liste des méta caractères ainsi que la manière de les échapper :

Méta caractères	Echappés en
?	\?
+	\+
.	\.
*	*
{	\{
|	\\|
(	\(
)	\)

IV - Utilisation

Dans les exemples qui vont suivrent, nous utiliserons l'expression suivante :

```
^www\.[-_[:alnum:]]+\.[[:lower:]]{2,4}
```

Cette expression régulière permet de reconnaître une adresse internet (un peu simplement puisque l'on se contente du nom de domaine).

Pour bien comprendre une expression, il faut l'analyser par morceaux :

```
^www\.
```

La chaîne doit commencer (^) par trois 'w' suivi d'un point (il faut bien sûr échapper le caractère '.'). on aurait trés bien pu écrire :

```
^[w]{3}\.
```

Beaucoup moins lisible.

```
[-_[:alnum:]]+
```

Nous avons à faire à un regroupement pour laisser plusieurs possibilités, la chaîne doit contenir :

- Le caractère '_'
- Le caractère '-'
- Un chiffre
- Une letrre majuscule
- Une letrre minuscule

Et ceux au moins une fois (dû au + après les crochets).

```
\.
```

Suivie d'un point.

```
[[:lower:]]{2,4}
```

Suivie de 2 à 4 lettres minuscules.

Il ne faut pas oublier que notre expression sera entre guillemets, il faut donc échapper les caractères spéciaux :

```
"www\\.[-_[:alnum:]]+\\.[[:lower:]]{2,4}"
```

IV-A - Savoir si une chaîne correspond à un motif

```
#include <stdio.h>
#include <stdlib.h>
#include <regex.h>
```

```c
int main (void)
{
   int err;
   regex_t preg;
   const char *str_request = "www.developpez.com";
   const char *str_regex = "www\\.[-_[:alnum:]]+\\.[[:lower:]]{2,4}";

/* (1) */
   err = regcomp (&preg, str_regex, REG_NOSUB | REG_EXTENDED);
   if (err == 0)
   {
      int match;

/* (2) */
      match = regexec (&preg, str_request, 0, NULL, 0);
/* (3) */
      regfree (&preg);
/* (4) */
      if (match == 0)
      {
         printf ("%s est une adresse internet valide\n", str_request);
      }
/* (5) */
      else if (match == REG_NOMATCH)
      {
         printf ("%s n\'est pas une adresse internet valide\n", str_request);
      }
/* (6) */
      else
      {
         char *text;
         size_t size;

/* (7) */
         size = regerror (err, &preg, NULL, 0);
         text = malloc (sizeof (*text) * size);
         if (text)
         {
/* (8) */
            regerror (err, &preg, text, size);
            fprintf (stderr, "%s\n", text);
            free (text);
         }
         else
         {
            fprintf (stderr, "Memoire insuffisante\n");
            exit (EXIT_FAILURE);
         }
      }
   }
   puts ("\nPress any key\n");
/* Dev-cpp */
   getchar ();
   return (EXIT_SUCCESS);
}
```

1 On commence par compiler notre expression régulière avec l'option REG_NOSUB pour vérifier uniquement la concordance
2 Si aucune erreur s'est produite on demande l'analyse de notre chaîne
3 On n'a plus besoin de l'expression compilée, on demande la libération de la mémoire
4 La chaîne est identifiée
5 On ne retrouve pas le motif de notre expression dans la chaîne analysée
6 Ou il s'est produite une erreur
7 Dans ce cas, on appel une première fois *regerror* pour connaître la taille du message d'erreur
8 Et une seconde fois pour récupérer la chaîne

IV-B - Extraire une sous chaîne correspondant à un motif

Toujours avec la même expression mais cette fois, on va récupérer les concordances. Vous avez peut être remarquer que la fonction *regexec*, lorsque l'on lui demande remplie un tableau de type *regmatch_t* dont la taille

est fournie par l'utilisateur. Mais comment connaître la taille du tableau? Cette information est contenue dans le membre *re_nsub* la structure *regex_t* remplie par *regcomp*, mais comment cette dernière peut connaître le nombre de motifs reconnus à l'avance? Tout simplement parce que chaque motif est entouré de parenthèses dans l'expression régulière, il lui suffit donc de compter le nombres de terme entre parenthèse. Il faut donc légèrement la modifier :

```
"(www\\.[-_[:alnum:]]+\\.[[:lower:]]{2,4})"
```

Et voici le code :

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

int main (void)
{
   int err;
   regex_t preg;
   const char *str_request = "http://www.developpez.net/forums/index.php";
   const char *str_regex = "(www\\.[-_[:alnum:]]+\\.[[:lower:]]{2,4})";

/* (1) */
   err = regcomp (&preg, str_regex, REG_EXTENDED);
   if (err == 0)
   {
      int match;
      size_t nmatch = 0;
      regmatch_t *pmatch = NULL;

      nmatch = preg.re_nsub;
      pmatch = malloc (sizeof (*pmatch) * nmatch);
      if (pmatch)
      {
/* (2) */
        match = regexec (&preg, str_request, nmatch, pmatch, 0);
/* (3) */
        regfree (&preg);
/* (4) */
        if (match == 0)
        {
           char *site = NULL;
           int start = pmatch[0].rm_so;
           int end = pmatch[0].rm_eo;
           size_t size = end - start;

           site = malloc (sizeof (*site) * (size + 1));
           if (site)
           {
              strncpy (site, &str_request[start], size);
              site[size] = '\0';
              printf ("%s\n", site);
              free (site);
           }
        }
/* (5) */
        else if (match == REG_NOMATCH)
        {
           printf ("%s n\'est pas une adresse internet valide\n", str_request);
        }
/* (6) */
        else
        {
           char *text;
           size_t size;

/* (7) */
           size = regerror (err, &preg, NULL, 0);
           text = malloc (sizeof (*text) * size);
           if (text)
           {
/* (8) */
              regerror (err, &preg, text, size);
```

```
                fprintf (stderr, "%s\n", text);
                free (text);
             }
             else
             {
                fprintf (stderr, "Memoire insuffisante\n");
                exit (EXIT_FAILURE);
             }
          }
       }
       else
       {
          fprintf (stderr, "Memoire insuffisante\n");
          exit (EXIT_FAILURE);
       }
    }
    puts ("\nPress any key\n");
/* Dev-cpp */
    getchar ();
    return (EXIT_SUCCESS);
}
```

1 On commence par compiler notre expression sans l'option REG_NOSUB pour pouvoir récupérer les sous chaîne reconnues
2 Grâce à la structure *regex_t*, on connaît le nombre de sous chaîne attendue, on alloue donc un tableau de taille adéquate pour les stocker
3 On demande l'analyse de notre chaîne
4 Si la chaîne est reconnue
5 On récupère les indices de début et de fin de la sous chaîne et on en profite pour calculer sa taille
6 On alloue un espace mémoire suffisant pour stocker notre sous chaîne
7 On copie la chaîne d'origine de l'indice *start* jusqu'à *size* caractères
8 Et on n'oublie pas de libérer la mémoire allouée

V - Conclusion

A présent, vous avez les connaissances nécessaires pour faire une bonne utilisation des expressions régulières en C. Si vous souhaitez en apprendre plus sur le sujet, vous pouvez consulter les pages man de la bibliothèque regex ou encore celle du programme grep; une documentation est aussi fournie avec la bibliothèque regex.

Toutefois, si vous souhaitez en faire une utilisation plus poussée, je vous conseille de vous tourner vers un langage tel que Perl.

VI - Autres documentations

Les expression régulières avec l'API Regex de Java par Hugo ETIEVANT.

Utilisation des expressions régulières en .Net par MORAND Louis-Guillaume.

VII - Remerciements

Merci à Gnux pour la relecture attentive de cet article.