

perldebtut

Table des matières

1	NAME/NOM	1
2	DESCRIPTION	1
3	Utiliser strict (use strict)	2
4	Observer les variables et -w et v	3
5	Aide (help)	4
6	Pas à pas dans le code	8
7	Emplacement réservé pour a, w, t, T	10
8	Expressions rationnelles (ou régulières)	11
9	Conseils sur les sorties	11
10	CGI	11
11	Interfaces graphiques (GUI)	12
12	Résumé	12
13	VOIR AUSSI	12
14	AUTEUR	12
15	CONTRIBUTIONS	12
16	TRADUCTION	12
	16.1 Version	12
	16.2 Traducteur	12
	16.3 Relecture	13
17	À propos de ce document	13

1 NAME/NOM

perldebtut - Tutoriel de débogage de Perl

2 DESCRIPTION

Une (très) légère introduction sur l'utilisation du débogueur de Perl, et un pointeur vers des sources d'information approfondies disponibles sur le sujet du débogage des programmes perl.

Il y a un nombre extraordinaire de personnes qui semblent ne rien connaître de l'utilisation de débogueur de Perl, bien qu'ils utilisent ce langage quotidiennement. Ceci est pour eux.

3 Utiliser strict (use strict)

Tout d'abord vous pouvez, sans utiliser le débogueur perl, mettre en oeuvre quelques idées pour vous simplifier la vie quand arrive le moment de déboguer des programmes perl. En guise de démonstration, voici un script simple, nommé "salut", présentant un problème :

```
#!/usr/bin/perl

$var1 = 'Salut le monde'; # toujours voulu faire cela :-)
$var2 = "$var1\n";

print $var2;
exit;
```

Bien que ce script se compile et s'exécute gaiement, il ne va probablement pas faire ce que l'on en attend, c'est à dire qu'il ne va pas du tout écrire "Salut le monde\n"; il va en revanche faire exactement ce que nous lui avons demandé de faire, les ordinateurs ayant légèrement tendance à agir de cette manière. Ce script va donc écrire un caractère de nouvelle ligne, et vous allez obtenir ce qui ressemble à une ligne blanche. Il semble que le script ne contiennent que deux variables, alors qu'il en contient en fait trois (à cause de la faute de frappe).

```
$var1 = 'Hello World';
$var1 = undef;
$var2 = "\n";
```

Afin de mettre en évidence ce type de problème, nous pouvons forcer la déclaration de chaque variable avant son utilisation en faisant appel au module «strict» par l'insertion après la première ligne du script de l'instruction 'use strict'. Maintenant, lors de l'exécution, perl se plaint de trois variables non déclarées, et nous obtenons quatre messages d'erreur car une variable est référencée deux fois :

```
Global symbol "$var1" requires explicit package name at ./t1 line 4.
Global symbol "$var2" requires explicit package name at ./t1 line 5.
Global symbol "$var1" requires explicit package name at ./t1 line 5.
Global symbol "$var2" requires explicit package name at ./t1 line 7.
Execution of ./hello aborted due to compilation errors.
```

Merveilleux ! Et pour supprimer ces erreurs, nous déclarons explicitement toutes les variables. Maintenant notre script ressemble à ceci :

```
#!/usr/bin/perl
use strict;

my $var1 = 'Salut le monde';
my $var1 = undef;
my $var2 = "$var1\n";

print $var2;
exit;
```

Nous procédons alors à une vérification de la syntaxe (toujours une bonne idée) avant d'exécuter à nouveau le script :

```
> perl -c hello
hello syntax OK
```

Et maintenant quand nous exécutons ce script, nous obtenons toujours "\n", mais au moins nous savons pourquoi. La compilation du script a révélé la variable '\$var1' (avec la lettre 'l') et le simple changement de \$var1 par \$varl résoud le problème.

4 Observer les variables et -w et v

D'accord, mais comment faire lorsque vous souhaitez réellement voir vos données, le contenu d'une variable dynamique, juste avant son utilisation ?

```
#!/usr/bin/perl
use strict;

my $key = 'welcome';
my %data = (
    'this' => qw(that),
    'tom' => qw(and jerry),
    'welcome' => q>Hello World),
    'zip' => q>welcome),
);
my @data = keys %data;

print "$data{$key}\n";
exit;
```

Tout semble correct. Après avoir été testé avec la vérification de syntaxe (**perl -c nom_du_script**) nous exécutons ce script et tout ce que nous obtenons est à nouveau une ligne blanche ! Hmmmmm. Dans ce cas, une approche courante du débogage serait de parsemer délibérément quelques instructions print, pour ajouter une première vérification juste avant d'écrire nos données, et une seconde juste après :

```
print "All OK\n" if grep($key, keys %data);
print "$data{$key}\n";
print "done: '$data{$key}'\n";
```

Après une nouvelle tentative :

```
> perl data
All OK

done: ''
```

Après s'être usé les yeux sur ce code pendant un bon moment, nous allons boire une tasse de café et nous tentons une nouvelle approche. C'est-à-dire que nous appelons la cavalerie en invoquant perl avec l'option **'-d'** sur la ligne de commande.

```
> perl -d data
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

main:./data:4):    my $key = 'welcome';
```

Donc, ce que nous venons de faire ici est de lancer notre script avec le débogueur intégré de perl. Celui-ci s'est arrêté à la première ligne de code exécutable et attend notre action.

Avant de poursuivre plus avant, vous allez souhaiter connaître comment quitter le débogueur : tapez simplement la lettre **'q'**, pas les mots **'quit'**, ni **'exit'**.

```
DB<1> q
>
```

C'est cela, vous êtes à nouveau au paddock.

5 Aide (help)

Démarrer à nouveau le débogueur et nous allons examiner le menu d'aide. Il y a plusieurs méthodes pour afficher l'aide: taper simplement '**h**' affichera une longue liste déroulante, '**|h**' (pipe-h) redirigera l'aide vers l'afficheur page par page ('more' ou 'less' probablement), et finalement, '**h h**' (h-espace-h) vous donnera un écran très utile résumant les commandes :

```
DB<1> h
List/search source lines:          Control script execution:
l [ln|sub] List source code        T          Stack trace
- or .      List previous/current line s [expr]   Single step [in expr]
v [line]    View around line       n [expr]   Next, steps over subs
f filename  View source in file    <CR/Enter> Repeat last n or s
/pattern/ ?patt? Search forw/backw  r          Return from subroutine
M          Show module versions    c [ln|sub] Continue until position
Debugger controls:                L          List break/watch/actions
o [...]    Set debugger options    t [expr]   Toggle trace [trace expr]
<[<|{|{|>|>] [cmd] Do pre/post-prompt b [ln|event|sub] [cmd] Set breakpoint
! [N|pat]  Redo a previous command  B ln|*     Delete a/all breakpoints
H [-num]   Display last num commands a [ln] cmd Do cmd before line
= [a val]  Define/list an alias     A ln|*     Delete a/all actions
h [db_cmd] Get help on command     w expr     Add a watch expression
h h        Complete help page       W expr|*   Delete a/all watch exprs
|[|]db_cmd Send output to pager     ![!] syscmd Run cmd in a subprocess
q or ^D    Quit                     R          Attempt a restart
Data Examination:      expr      Execute perl code, also see: s,n,t expr
x|m expr      Evals expr in list context, dumps the result or lists methods.
p expr        Print expression (uses script's current package).
S [!]pat]    List subroutine names [not] matching pattern
V [Pk [Vars]] List Variables in Package. Vars can be ~pattern or !pattern.
X [Vars]     Same as "V current_package [Vars]". i class inheritance tree.
y [n [Vars]] List lexicals in higher scope <n>. Vars same as V.
e           Display thread id      E Display all thread ids.
For more help, type h cmd_letter, or run man perldebug for all docs.
```

Plus d'options que vous ne le pensiez ! Ce n'est pas si terrible qu'il y paraît, et il est très utile et en plus amusant, d'en connaître plus à propos de chacune de ces commandes.

Il y a quelques options utiles à connaître immédiatement. Vous pensez probablement que vous n'utilisez aucune bibliothèque en ce moment, mais '**M**' va vous montrer les modules actuellement actifs, ceux appelés aussi bien par le débogueur que par le script. De même, '**m**' vous affichera les méthodes et '**S**' tous les sous-programmes (par motif) comme ci-dessous. '**V**' et '**X**' montrent les variables du programme rangées par package et peuvent être restreintes par l'utilisation de motifs.

```
DB<2>S str
dumpvar::stringify
strict::bits
strict::import
strict::unimport
```

L'utilisation de '**X**' et de ses cousins demande de ne pas utiliser l'identificateur de type (\$@%), seulement le 'nom' :

```
DM<3>X ~err
FileHandle(stderr) => fileno(2)
```

Rappelez-vous que nous sommes dans notre minuscule programme avec un problème, nous devrions jeter un oeil pour voir où nous sommes et à quoi nos données ressemblent. Avant tout, regardons un peu de code autour de notre position actuelle (la première ligne de code dans notre cas), grâce à la lettre '**v**' :

```
DB<4> v
1      #!/usr/bin/perl
2:     use strict;
3
```

```

4==>   my $key = 'welcome';
5:      my %data = (
6          'this' => qw(that),
7          'tom'  => qw(and jerry),
8          'welcome' => q(Hello World),
9          'zip'  => q(welcome),
10         );

```

À la ligne 4 se présente un pointeur utile, qui vous indique où vous vous situez actuellement. Pour voir plus avant dans le programme, tapons encore 'v'.

```

DB<4> w
8          'welcome' => q(Hello World),
9          'zip'    => q(welcome),
10         );
11:      my @data = keys %data;
12:      print "All OK\n" if grep($key, keys %data);
13:      print "$data{$key}\n";
14:      print "done: '$data{$key}'\n";
15:      exit;

```

Et si vous voulez lister à nouveau la ligne 5, tapez 'l 5', (remarquez l'espace):

```

DB<4> l 5
5:      my %data = (

```

Dans le cas de ce script, il n'y pas grand chose à voir, mais il peut y avoir des pages entières à explorer et 'l' peut être très utile. Pour réinitialiser votre vue sur la ligne qui va s'exécuter, tapez un simple point '.':

```

DB<5> .
main:./data_a:4):      my $key = 'welcome';

```

La ligne affichée est la prochaine qui sera exécutée, sans que cette exécution ne se soit produite pour l'instant. Donc bien que nous puissions imprimer une variable grâce à la commande 'p', tout ce que nous allons obtenir est une valeur vide (non définie). Ce que nous devons faire est exécuter la prochaine instruction exécutable en tapant 's':

```

DB<6> s
main:./data_a:5):      my %data = (
main:./data_a:6):          'this' => qw(that),
main:./data_a:7):          'tom'  => qw(and jerry),
main:./data_a:8):          'welcome' => q(Hello World),
main:./data_a:9):          'zip'  => q(welcome),
main:./data_a:10):      );

```

Maintenant, nous pouvons observer cette première variable (\$clef)

```

DB<7> p $key
welcome

```

La ligne 13 est celle où l'action a lieu, donc continuons notre progression grâce à la lettre 'c', qui insert un point d'arrêt 'à usage unique' à la ligne ou au sous-programme indiqué.

```

DB<8> c 13
All OK
main:./data_a:13):      print "$data{$key}\n";

```

Nous avons dépassé notre vérification (où 'All OK' était imprimé) et nous nous sommes arrêtés juste avant le gros du travail. Nous pourrions essayer de lister quelques variables pour voir ce qu'il se passe.

```

DB<9> p $data{$key}

```

Pas grand chose ici, observons notre hachage :

```
DB<10> p %data
Hello Worldziptomandwelcomejerrywelcomethisthat

DB<11> p keys %data
Hello Worldtomwelcomejerrythis
```

Bon, ceci n'est pas très facile à lire, et dans l'écran d'aide (**h h**), la commande '**x**' semble prometteuse :

```
DB<12> x %data
0 'Hello World'
1 'zip'
2 'tom'
3 'and'
4 'welcome'
5 undef
6 'jerry'
7 'welcome'
8 'this'
9 'that'
```

Ceci n'aide pas beaucoup, deux 'bienvenue', mais sans que l'on sache s'ils correspondent à des clefs ou à des valeurs. Il s'agit juste de l'affichage d'une image d'un tableau, dans ce cas sans grande utilité. L'astuce est ici d'utiliser une référence à la structure de données :

```
DB<13> x \%data
0 HASH(0x8194bc4)
  'Hello World' => 'zip'
  'jerry' => 'welcome'
  'this' => 'that'
  'tom' => 'and'
  'welcome' => undef
```

La référence est réellement détaillée, et nous mettons finalement en évidence notre problème. Notre citation est parfaitement valide, mais erronée pour notre propos, avec 'and jerry' traité comme deux mots séparés plutôt que comme une phrase, donc entraînant un mauvais alignement des paires du hachage.

Le commutateur '**-w**' nous aurait indiqué ce problème si nous l'avions utilisé au départ, et nous aurait évité pas mal de soucis :

```
> perl -w data
Odd number of elements in hash assignment at ./data line 5.
```

Nous réparons notre citation : 'tom' =>q(et jerry), et exécutons à nouveau notre programme. Nous obtenons alors le résultat attendu :

```
> perl -w data
Bonjour le monde
```

Pendant que nous y sommes, regardons de plus près la commande '**x**'. Elle est réellement utile et vous affichera merveilleusement le contenu de références imbriquées, d'objet complets, de fragments d'objets, de tout ce que vous voudrez bien lui appliquer :

Construisons rapidement un objet, et e-x-plorons le. D'abord, démarrons le débogueur : il demande une entrée à partir de STDIN, donnons-lui donc quelque chose qui n'engage à rien, un zéro :

```
> perl -de 0
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.
```

```
Enter h or `h h` for help, or `man perldebug` for more help.
```

```
main::(-e:1): 0
```

Maintenant construisez au vol un objet sur quelques lignes (notez l'antislash):

```
DB<1> $obj = bless({'unique_id'=>'123', 'attr'=> \
cont: {'col' => 'black', 'things' => [qw(this that etc)]}, 'MY_class')
```

Et jetez un oeil dessus:

```
DB<2> x $obj
0 MY_class=HASH(0x828ad98)
  'attr' => HASH(0x828ad68)
'col' => 'black'
'things' => ARRAY(0x828abb8)
  0 'this'
  1 'that'
  2 'etc'
  'unique_id' => 123
DB<3>
```

Utile, n'est-ce pas? Vous pouvez utiliser eval sur n'importe quoi, et expérimenter avec des fragments de programme ou d'expressions régulières jusqu'à ce que les vaches rentrent à l'étable:

```
DB<3> @data = qw(this that the other atheism leather theory scythe)

DB<4> p 'saw -> ' . ($cnt += map { print "\t:\t$_\n" } grep(/the/, sort @data))
atheism
leather
other
scythe
the
theory
saw -> 6
```

Si vous voulez voir l'historique des commandes, tapez un 'H':

```
DB<5> H
4: p 'saw -> ' . ($cnt += map { print "\t:\t$_\n" } grep(/the/, sort @data))
3: @data = qw(this that the other atheism leather theory scythe)
2: x $obj
1: $obj = bless({'unique_id'=>'123', 'attr'=>
{'col' => 'black', 'things' => [qw(this that etc)]}, 'MY_class')
DB<5>
```

Et si vous voulez répéter une quelques commande précédente, utilisez le point d'exclamation: '!':

```
DB<5> !4
p 'saw -> ' . ($cnt += map { print "$_\n" } grep(/the/, sort @data))
atheism
leather
other
scythe
the
theory
saw -> 12
```

Pour plus d'informations sur les références, voyez les pages de manuel perlref et perlreftut.

6 Pas à pas dans le code

Voici un simple programme qui converti les degrés Fahrenheit en Celsius (et vice-versa). Ce programme présente aussi un problème.

```
#!/usr/bin/perl -w
use strict;

my $arg = $ARGV[0] || '-c20';

if ($arg =~ /^-(c|f)((\-|\+)*\d+(\.\d+)*$/) {
    my ($deg, $num) = ($1, $2);
    my ($in, $out) = ($num, $num);
    if ($deg eq 'c') {
        $deg = 'f';
        $out = &c2f($num);
    } else {
        $deg = 'c';
        $out = &f2c($num);
    }
    $out = sprintf('%0.2f', $out);
    $out =~ s/^\((\-|\+)*\d+\)\.0+$/\1/;
    print "$out $deg\n";
} else {
    print "Usage: $0 -[c|f] num\n";
}
exit;

sub f2c {
    my $f = shift;
    my $c = 5 * $f - 32 / 9;
    return $c;
}

sub c2f {
    my $c = shift;
    my $f = 9 * $c / 5 + 32;
    return $f;
}
```

Pour une raison inconnue, la conversion des Fahrenheit en Celsius ne donne pas le résultat escompté. Voici ce que ce programme effectue :

```
> temp -c0.72
33.30 f

> temp -f33.3
162.94 c
```

Pas très cohérent ! Nous allons placer manuellement un point d'arrêt dans le programme et exécuter celui-ci dans le débogueur pour voir ce qui se passe. Un point d'arrêt est un drapeau. Le débogueur va exécuter le programme sans interruption jusqu'à ce drapeau. Quand il l'atteint, il arrête l'exécution et présente une invite pour la suite de l'interaction. En utilisation normale, ces commandes de débogage sont totalement ignorées, et elle peuvent être conservées en toute sécurité dans le programme final, bien qu'elles en augmentent quelque peu le désordre.

```
my ($in, $out) = ($num, $num);
$DB::single=2; # insert at line 9!
if ($deg eq 'c')
    ...
```

```
> perl -d temp -f33.3
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

main::(temp:4): my $arg = $ARGV[0] || '-c20';
```

Nous allons simplement descendre jusqu'à notre point d'arrêt prédéfini grâce à la commande 'c' :

```
DB<1> c
main::(temp:10):                if ($deg eq 'c') {
```

suivie par une commande 'v' pour voir où nous sommes :

```
DB<1> v
7:          my ($deg, $num) = ($1, $2);
8:          my ($in, $out) = ($num, $num);
9:          $DB::single=2;
10==>      if ($deg eq 'c') {
11:                $deg = 'f';
12:                $out = &c2f($num);
13          } else {
14:                $deg = 'c';
15:                $out = &f2c($num);
16          }
```

et par une commande 'p' pour montrer le contenu des variables que nous utilisons :

```
DB<1> p $deg, $num
f33.3
```

Nous pouvons placer un autre point d'arrêt à n'importe quelle ligne commençant par un numéro suivi par deux points. Nous utiliserons la ligne 17, cette ligne étant la première après la sortie du sous-programme f2c. Nous prévoyons de faire une pause à cet endroit plus tard.

```
DB<2> b 17
```

Il n'y a pas d'indications en retour, mais vous pouvez voir les points d'arrêt placés en utilisant la commande 'L'.

```
DB<3> L
temp:
17:          print "$out $deg\n";
          break if (1)
```

Il est à noter que les points d'arrêts peuvent être supprimés en utilisant les commandes 'd' ou 'D'.

Nous allons alors continuer plus avant dans notre sous-programme, cette fois en utilisant, à la place du numéro de ligne, le nom du sous-programme suivi par la commande 'v', maintenant familière,

```
DB<3> c f2c
main::f2c(temp:30):                my $f = shift;

DB<4> v
24:      exit;
25
26      sub f2c {
27==>          my $f = shift;
28:          my $c = 5 * $f - 32 / 9;
29:          return $c;
30      }
31
32      sub c2f {
33:          my $c = shift;
```

Notez que si il existait un appel à un sous-programme entre notre position et la ligne 29, et si nous voulions traverser ce sous-programme pas-à-pas, nous pourrions utiliser la commande 's'. Pour le franchir, il faudrait utiliser 'n' qui exécuterait alors le sous-programme sans l'inspecter. Dans notre cas, nous continuons simplement jusqu'à la ligne 29.

Regardons la valeur retournée :

```
DB<5> p $c
162.94444444444444
```

Ce n'est pas du tout la réponse exacte, mais l'opération semble correcte. Je me demande si le problème ne serait pas lié aux priorités des opérateurs ? Essayons notre opération avec quelques autres combinaisons de parenthèses.

```
DB<6> p (5 * $f - 32 / 9)
162.94444444444444
```

```
DB<7> p 5 * $f - (32 / 9)
162.94444444444444
```

```
DB<8> p (5 * $f) - 32 / 9
162.94444444444444
```

```
DB<9> p 5 * ($f - 32) / 9
0.7222222222222221
```

:-) Ce dernier essai ressemble plus à ce que l'on souhaite ! Bien, maintenant nous pouvons définir la variable à renvoyer, et nous allons sortir de notre sous-programme grâce à la commande 'r' :

```
DB<10> $c = 5 * ($f - 32) / 9
```

```
DB<11> r
scalar context return from main::f2c: 0.7222222222222221
```

Semble correct, continuons donc jusqu'à la fin du script :

```
DB<12> c
0.72 c
Debugged program terminated. Use q to quit or R to restart,
use O inhibit_exit to avoid stopping after program termination,
h q, h R or h O to get additional info.
```

Une correction rapide de la ligne erronée (insertion des parenthèses manquantes) dans le programme et nous avons terminé.

7 Emplacement réservé pour a, w, t, T

Action, observation des variables, suivi des piles etc...: sur la liste des choses à faire.

a
w
t
T

8 Expressions rationnelles (ou régulières)

Avez-vous jamais voulu savoir à quoi ressemble une expression régulière ? Vous aurez besoin pour cela de compiler Perl avec le drapeau `DEBUGGING` :

```
> perl -Dr -e '/^pe(a)*rl$/i'
Compiling REx '^pe(a)*rl$'
size 17 first at 2
rarest char
at 0
  1: BOL(2)
  2: EXACTF (4)
  4: CURLYN[1] {0,32767}(14)
  6:  NOTHING(8)
  8:  EXACTF (0)
 12:  WHILEM(0)
 13:  NOTHING(14)
 14:  EXACTF (16)
 16:  EOL(17)
 17:  END(0)
floating '$ at 4..2147483647 (checking floating) stclass 'EXACTF '
anchored(BOL) minlen 4
Omitting '$ $& $' support.

EXECUTING...

Freeing REx: '^pe(a)*rl$'
```

Voulez-vous vraiment savoir?:-) Pour plus de détails sanglants sur la mise en oeuvre des expressions régulières, regardez les pages de manuel `perlre`, `perlretut`, et pour décoder les étiquettes mystérieuses (voir ci-dessus `BOL`, `CURLYN` etc..), se référer au manuel `perldebug`.

9 Conseils sur les sorties

Pour obtenir toutes les sorties de votre journal d'erreur, et ne manquer aucun messages via les tampons du système d'exploitation, insérez au début de votre script une ligne de ce type :

```
$|=1;
```

Pour voir l'extrémité d'un fichier journal en croissance dynamique, tapez à partir de la ligne de commande :

```
tail -f $error_log
```

Envelopper tous les appels à `'die'` dans une routine peut être utilisé pour voir comment et à partir de quoi ils sont appelés. La page de manuel `perlvar` présente plus d'informations :

```
BEGIN { $SIG{__DIE__} = sub { require Carp; Carp::confess(@_) } }
```

Diverses techniques utilisées pour la redirection des descripteurs `STDOUT` et `STDERR` sont expliquées dans les pages de manuel `perloutent` et `perlfaq8`.

10 CGI

Juste un petit truc pour tous les programmeurs CGI qui ne trouvent comment poursuivre après l'invite `'waiting for input'` quand ils exécutent leur script à partir de la ligne de commande. Essayez quelque chose comme :

```
> perl -d my_cgi.pl -nodebug
```

Bien sûr les pages de manuel CGI et `perlfaq9` vous en dirons plus.

11 Interfaces graphiques (GUI)

L'interface de la ligne de commande est étroitement intégrée avec une extension à emacs et il existe aussi une interface vers vi.

Vous n'avez pas à faire tout ceci à partir de la ligne de commande, il existe quelques possibilités d'utilisation d'interfaces graphiques. Ce qui est agréable alors est de pouvoir agiter la souris au dessus d'une variable et de voir apparaître dans une fenêtre appropriée ou dans un ballon d'aide son contenu. Plus besoin de se fatiguer à taper '**x \$nomdevariable**' :-)

En particulier, vous pouvez partir à la chasse de :

ptkdb Enveloppe pour le débogueur intégré basée sur perlTK

ddd data display debugger (débogueur exposeur de données)

PerlDevKit et **PerlBuilder** sont des outils spécifiques à Windows NT.

NB. Plus d'informations sur ces outils ainsi que sur d'autres équivalents seraient appréciées.

12 Résumé

Nous avons vu comment encourager de bonnes pratiques de programmation grâce à l'utilisation de **use strict** et de **-w**. Nous pouvons exécuter le débogueur de perl (**perl -d nomdescript**) pour inspecter nos données grâce aux commandes **p** et **x**. Vous pouvez parcourir votre code, placer des points d'arrêt avec la commande **b**, parcourir ce programme pas à pas avec **s** ou **n**, continuer avec **c** et revenir d'un sous-programme avec **r**. Plutôt intuitif quand on y regarde de près.

Il y a bien sûr beaucoup plus à découvrir à propos du débogueur, ce tutoriel n'ayant fait qu'en explorer la surface. La meilleure façon d'en apprendre plus est d'utiliser perldoc pour découvrir beaucoup d'autres d'aspects du langage, de lire l'aide en ligne (la page de manuel perlbug devrait être votre prochaine lecture) et bien sûr d'expérimenter.

13 VOIR AUSSI

Les pages de manuel perldebug, perldebguts, perldiag, perlrun, *dprofpp*.

14 AUTEUR

Richard Foley <richard@rfi.net> Copyright(c) 2000

15 CONTRIBUTIONS

Diverses personnes ont aidées par leurs suggestions et leurs contributions, en particulier :

Ronald J Kimball <jk@linguist.dartmouth.edu>

Hugo van der Sanden <hv@crypt0.demon.co.uk>

Peter Scott <Peter@PSDT.com>

16 TRADUCTION

16.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

16.2 Traducteur

Traduction initiale : Landry Le Chevanton <landry.lechevanton@wanadoo.fr>. Mise à jour : Paul Gaborit <paul.gaborit@enstimac.fr>.

16.3 Relecture

G rard Delafond

17   propos de ce document

Ce document est la traduction franaise du document original distribu  avec perl. Vous pouvez retrouver l'ensemble de la documentation franaise Perl ( ventuellement mise   jour) en consultant l'URL <<http://perl.enstimac.fr/>>.

Ce document PDF a  t  produit Paul Gaborit. Si vous utilisez la version PDF de cette documentation (ou une version papier issue de la version PDF) pour tout autre usage qu'un usage personnel, je vous serai reconnaissant de m'en informer par un petit message <<mailto:Paul.Gaborit@enstimac.fr>>.

Si vous avez des remarques concernant ce document, en premier lieu, contactez la traducteur (vous devriez trouver son adresse  lectronique dans la rubrique TRADUCTION) et expliquez-lui gentiment vos remarques ou critiques. Il devrait normalement vous r pondre et prendre en compte votre avis. En l'absence de r ponse, vous pouvez  ventuellement me contacter.

Vous pouvez aussi participer   l'effort de traduction de la documentation Perl. Toutes les bonnes volont s sont les bienvenues. Vous devriez trouver tous les renseignements n cessaires en consultant l'URL ci-dessus.

Ce document PDF est distribu  selon les termes de la license Artistique de Perl. Toute autre distribution de ce fichier ou de ses d riv s impose qu'un arrangement soit fait avec le(s) propri taire(s) des droits. Ces droits appartiennent aux auteurs du document original (lorsqu'ils sont identifi s dans la rubrique AUTEUR), aux traducteurs et relecteurs pour la version franaise et   moi-m me pour la version PDF.