perlrun

Table des matières

1 NAME/NOM

perlrun - Comment utiliser l'interpréteur Perl

2 SYNOPSIS

perl [**-CsTuUWX**] [**-hv**] [**-V**[:*configvar*]] [**-cw**] [**-d**[:*debugger*]] [**-D**[*number/list*]] [**-pna**] [**-F***pattern*] [**-l**[*octal*]] [**-0**[*octal/hexadecimal*]] [**-I***dir*] [**-m**[**-**]*module*] [**-M**[**-**]*'module...'*] [**-f**] [**-C [*number/list*]**] [**-P**] [**-S**] [**-x**[*dir*]] [**-i**[*extension*]] [**-e** *'command'*] [**–**] [*programfile*] [*argument*]...

3 DESCRIPTION

La façon habituelle d'exécuter un programme Perl est de le rendre directement exécutable ou bien de passer le nom du fichier source comme argument de ligne de commande (Un environnement Perl interactif est aussi possible – voir *perldebug* pour plus de détails sur son utilisation). Au lancement, Perl recherche votre script dans l'un des endroits suivants :

1. Spécifié ligne par ligne par l'intermédiaire d'options **-e** sur la ligne de commande.
2. Contenu dans le fichier spécifié par le premier nom de fichier sur la ligne de commande. (Notez que les systèmes qui supportent la notation #! invoquent les interpréteurs de cette manière. Cf. Emplacement de Perl.)
3. Passé implicitement sur l'entrée standard. Cela ne marche que s'il n'y a pas de noms de fichiers comme argument – pour passer des arguments à un programme lisant sur STDIN, vous devez explicitement spécifier un "-" pour le nom du script.

Avec les méthodes 2 et 3, Perl démarre l'analyse du fichier d'entrée à partir du début, à moins de rajouter une option **-x**. Dans ce cas, il cherche la première ligne commençant par #! et contenant le mot "perl". L'interprétation commence alors à partir de cette ligne. C'est utile pour lancer un programme contenu dans un message plus grand. (Dans ce cas, la fin du programme doit être indiquée par `__END__`.)

Lorsque la ligne commençant par #! est analysée, les éventuelles options sont toujours recherchées. Ainsi, si vous êtes sur une machine qui n'autorise qu'un seul argument avec la ligne #!, ou pire, qui ne reconnait même pas la ligne #!, vous pouvez toujours obtenir un comportement homogène vis-à-vis des options, quelle que soit la manière dont Perl a été invoqué, même si **-x** a été utilisé pour trouver le début du programme.

Puisque historiquement certains systèmes d'exploitation arrêtaient l'interprétation de la ligne #! par le noyau à partir de 32 caractères, certaines options peuvent être passées sur la ligne de commande, d'autres pas ; vous pouvez même vous retrouver avec un "-" sans sa lettre au lieu d'une option complète, si vous n'y faites pas attention. Vous devrez probablement vous assurer que toutes vos options tombent d'un côté ou de l'autre de la frontière des 32 caractères. La plupart des options ne se soucient pas d'être traitées de façon redondante, mais obtenir un "-" au lieu d'une option complète pourrait pousser Perl à essayer d'exécuter l'entrée standard au lieu de votre programme. Une option **-I** incomplète pourrait aussi donner des résultats étranges.

Certaines options sont sensibles au nombre de fois où elles sont présentes sur la ligne de commande, par exemple, les combinaisons des options **-l** et **-0**. Il faut soit mettre toutes les options après la limite des 32 caractères (si possible), soit remplacer les utilisations de **-0***digits* par `BEGIN{ $/ = "\0digits"; }`.

L'analyse des options commence dès que "perl" est mentionné sur la ligne. Les séquences "-*" et "- " sont spécialement ignorées de telle manière que vous puissiez écrire (si l'envie vous en prend), disons

```
#!/bin/sh -- # -*- perl -*- -p
eval 'exec perl -wS $0 ${1+"$@"}'
    if $running_under_some_shell;
```

pour permettre à Perl de voir l'option **-p**.

Un truc similaire implique le programme **env**, si vous le possédez.

```
#!/usr/bin/env perl
```

Les exemples ci-dessus utilise un chemin relatif vers l'interpréteur perl et utilise donc la première version trouvée dans le PATH de l'utilisateur. Si vous voulez une version spécifique de Perl, disons, perl5.005_57, vous devez le placer directement dans le chemin de la ligne #! .

Si la ligne #! ne contient pas le mot "perl", le programme indiqué après le #! est lancé au lieu de l'interpréteur Perl. C'est assez bizarre, mais cela aide les gens qui utilisent des machines qui ne reconnnaissent pas #!, car ainsi ils peuvent dire à un programme que leur SHELL est */usr/bin/perl*, et Perl enverra le programme vers le bon interpréteur pour eux.

Après avoir localisé votre programme, Perl le compile en une forme interne. Si il y a ne serait-ce qu'un erreur de compilation, l'exécution du programme n'est pas tentée (c'est différent d'un script shell classique, qui peut être en partie exécuté avant que soit détectée une erreur de syntaxe).

Si le programme est syntaxiquement correct, il est exécuté. Si le programme se termine sans rencontrer un opérateur die() ou exit(), un `exit(0)` implicite est ajouté pour indiquer une exécution réussie.

3.1 #! et l'isolement (quoting) sur les systèmes non-Unix

La technique #! d'Unix peut être simulée sur les autres systèmes :

OS/2
: Ajoutez

```
extproc perl -S -vos_options
```

sur la première ligne de vos fichiers `*.cmd` (le **-S** est dû à un bug dans la manière dont cmd.exe gère 'extproc').

MS-DOS
: Créez un fichier de commande (batch) pour lancer votre programme, et placez son nom dans `ALTERNAT_SHEBANG` ("#!" se prononce She Bang, cf. Jargon File, NDT). Pour plus d'informations à ce sujet, voir le fichier *dosish.h* dans les sources de la distribution.

Win95/NT
: L'installation Win95/NT, en tout cas l'installeur Activestate pour Perl, va modifier la "base de registre" pour associer l'extension *.pl* avec l'interpréteur Perl. Si vous installez Perl par d'autres moyens (y compris via une compilation à partir des sources), vous devrez modifier la base de registre vous-même. Notez que cela implique que vous ne pourrez plus faire la différence entre un exécutable Perl et une bibliothèque Perl.

Macintosh
: Dans MacOS "Classic", les programmes Perl auront le Creator et le Type adéquat, donc un double-clic lancera l'application MacPerl. Dans Mac OS X, un script avec `#!` est transformable en une application double-cliquable en utilisant l'utilitaire DropScript de Wil Sanchez : http://www.wsanchez.net/software/ .

VMS

Ajoutez

```
$ perl -mysw 'f$env("procedure")' 'p1' 'p2' 'p3' 'p4' 'p5' 'p6' 'p7' 'p8' !
$ exit++ + ++$status != 0 and $exit = $status = undef;
```

au début de votre programme, où `-mysw` représente toutes les options de ligne de commande que vous voulez passer à Perl. Vous pouvez désormais invoquer le programme directement, en disant `perl programme`, ou en tant que procédure DCL, en disant `@programme` (ou implicitement via *DCL$PATH* en utilisant juste le nom du programme). Cette incantation est un peu difficile à mémoriser, mais Perl l'affichera pour vous si vous dites `perl "-V:startperl"`.

Les interpréteurs de commandes des systèmes non-Unix ont des idées plutôt différentes des shells Unix concernant l'isolement (quoting). Il vous faudra apprendre quels sont les caractères spéciaux de votre interpréteur de commandes (`*`, `\` et `"` le sont couramment), et comment protéger les espaces et ces caractères pour lancer des one-liners [Ndt] scripts sur une seule ligne [Fin de Ndt] (cf. **-e** plus bas).

Sur certains systèmes, vous devrez peut-être changer les apostrophes (single-quotes) en guillemets (double quotes), ce qu'il ne faut *pas* faire sur les systèmes Unix ou Plan9. Vous devrez peut-être aussi changer le signe % seul en %%.

Par exemple :

```
# Unix
perl -e 'print "Hello world\n"'

# MS-DOS, etc.
perl -e "print \"Hello world\n\""

# Macintosh
print "Hello world\n"
 (puis Run "Myscript" ou Shift-Command-R)

# VMS
perl -e "print ""Hello world\n"""
```

Le problème, c'est que le fonctionnement de ces exemples n'est absolument pas garanti : cela dépend de la commande et il est possible qu'aucune forme ne fonctionne. Si l'interpréteur de commande était **4DOS**, ceci marcherait probablement s<mieux :>

```
perl -e "print <Ctrl-x>"Hello world\n<Ctrl-x>""
```

CMD.EXE dans Windows NT a récupéré beaucoup de fonctionnalités Unix lorsque tout le monde avait le dos tourné, mais essayez de trouver de la documentation sur ses règles d'isolement !

Sur Macintosh, cela dépend de l'environnement que vous utilisez. Le shell MacPerl, ou MPW, est très proche des shells Unix pour son support de différentes variantes d'isolement, si ce n'est qu'il utilise allègrement les caractères Macintosh non ASCII comme caractères de contrôle.

Il n'y a pas de solution générale à ces problèmes. C'est juste le foutoir.

3.2 Emplacement de Perl

Cela peut sembler évident, mais Perl est utile lorsque les utilisateurs y ont facilement accès. Autant que possible, il est bon d'avoir */usr/bin/perl* et */usr/local/bin/perl* comme liens symboliques (symlinks) vers le bon binaire. Si cela n'est pas possible, nous encourageons les administrateurs systèmes à mettre perl et les utilitaires associés (directement ou sous forme de liens symboliques) dans un répertoire présent dans le PATH des utilisateurs, ou encore dans un quelconque autre endroit pratique et évident.

Dans cette documentation, `#!/usr/bin/perl` dans la première ligne d'un programme, indiquera ce qui marche sur votre système. Nous vous conseillons d'utiliser un chemin spécifique si vous avez besoin d'une version particulière.

```
#!/usr/local/bin/perl5.00554
```

ou si vous voulez juste utiliser une version supérieure ou égale à une version particulière, placez une instruction telle que celle-ci au début de votre programme :

```
use 5.005_54;
```

3.3 Options de ligne de commandes

Comme pour toutes les commandes standard, une option mono-caractère peut être combinée avec l'option suivante, le cas échéant.

```
#!/usr/bin/perl -spi.orig   # équivalent à -s -p -i.orig
```

Les options comprennent :

-0[*octal/hexadecimal*]

indique le séparateur d'enregistrments en entrée (`$/`) en notation octale ou hexadécimale. S'il n'y a pas de chiffres, le caractère nul (ASCII 0) est le séparateur. D'autres options peuvent suivre ou précéder les chiffres. Par exemple, si vous avez une version de **find** qui peut afficher les noms de fichiers terminés par des caratères nuls, vous pouvez écrire ceci :

```
find . -name '*.orig' -print0 | perl -n0e unlink
```

La valeur spéciale 00 va indiquer à Perl d'avaler les fichiers en mode paragraphes. La valeur 0777 indique à Perl d'avaler les fichiers en entier car il n'y a pas de caractères avec cette valeur octale.

Si vous voulez spécifier un caractère Unicode, utilisez la notation hexadécimale : `-0xHHH...`, où les `H` représentent des chiffres hexadécimaux. (Cela signifie que vous ne pouvez pas utiliser l'option `-x` avec un nom de répertoire constituée uniquement de chiffres hexadécimaux.)

-a

active le mode auto-découpage (autosplit) lorsqu'utilisé avec **-n** ou **-p**. On insère une commande split implicite vers le tableau @F au début de la boucle while implicite produite par **-n** ou **-p**.

```
perl -ane 'print pop(@F), "\n";'
```

est équivalent à :

```
while (<>) {
    @F = split(' ');
    print pop(@F), "\n";
}
```

Un autre séparateur [Ndt] que espace ' '[Fin Ndt] peut être spécifié via **-F**.

-C [*nombre/liste*]

L'option `-C` contrôle les fonctionnalités Unicode de Perl.

Depuis la version 5.8.1, le `-C` peut être suivi par un nombre ou une suite de lettres. Les lettres, leur valeur numérique et leur effet sont les suivants (une liste de lettre est équivalente à la somme de leur valeur) :

```
I     1    STDIN est supposé être en UTF-8
O     2    STDOUT sera en UTF-8
E     4    STDERR sera en UTF-8
S     7    I + O + E
i     8    UTF-8 est le filtre PerlIO par défaut pour les flux entrants
o    16    UTF-8 est le filtre PerlIO par défaut pour les flux sortants
D    24    i + o
A    32    les élements de @ARGV sont des chaînes codées en UTF-8
L    64    normalement "IOEioA" s'appliquent sans condition.
           Le L conditionne leur prise en compte aux variables d'environnement
           (LC_ALL, LC_TYPE et LANG, par ordre de priorité décroissant).
           Si ces variables indique UTF-8 alors les options "IOEioA"
           sont prises en compte.
```

Par exemple, `-COE` ou `-C6` activent l'UTF-8 sur STDTOUT et STDERR. Des lettres répétées sont juste redondantes. Elles ne se cumulent pas ou ne s'annulent pas.

Les options `io` indique que tous les open() (ou autres opérations d'E/S similaires) utiliseront implicitement le filtre PerlIO `:utf8`. En d'autres termes, tous les flux entrants seront lus en UTF-8 et tous les flux sortants seront écrits en UTF-8. Ce n'est que le comportement par défaut puisqu'il est toujours possible de changer ce comportement en indiquant explicitement les filtres voulus dans les appels à open() ou à binmode().

L'option `-C` seule ou la chaîne vide `""` pour la variable d'environnement `PERL_UNICODE` ont le même effet que `-CSDL`. En d'autres termes, les flux d'E/S standard et le filtre par défaut de `open()` sont UTF-8 **uniquement** si les

variables d'environnement demandent un locale UTF-8. Ce comportement reproduit le comportement implicite (en problématique) de Perl 5.8.0 vis-à-vis d'UTF-8.

Vous pouvez utiliser `-C0` (ou `"0"` pour `PERL_UNICODE`) pour déactiver explicitement toutes les fonctionnalités Unicode ci-dessus.

La variable magique `${^UNICODE}`, accessible uniquement en lecture, reflète la valeur numérique de ces réglages. Cette variable est calculée lors du démarrage de Perl et n'est plus modifiable par la suite. Si vous voulez changer de comportement dynamiquement, utilisez open() avec trois arguments (voir open in *perlfunc*), binmode() avec deux arguments (voir binmode in *perlfunc*) ou la directive `open` (voir *open*).

(Dans les versions de Perl antérieures à la version 5.8.1, l'option `-C` n'existait que sur Win32 et permettait d'activer les "wide system call" Unicode de l'API Win32. Cette fonctionnalité n'était quasiment pas utilisée et l'option a donc été "recyclée".)

-c

demande à Perl de vérifier la syntaxe du programme puis de quitter sans l'exécuter. En fait, il *va* exécuter les blocs `BEGIN` et `CHECK` ainsi que les instructions `use` car ils sont considérés comme se déroulant avant l'exécution de votre programme. En revanche, les blocs `INIT` et `END` ne seront pas exécutés.

-d

-dt

lance le programme sous le débogueur (debugger) Perl. Cf. *perldebug*. Si le **t** est présent, cela indique au débogueur que les fils d'exécution (les threads) sont utilisés dans le code à déboguer.

-d:*foo[=bar,baz]*

-dt:*foo[=bar,baz]*

lance le programme sous le contrôle du module de traçabilité, de profilage ou de débogage installé sous le nom Devel::foo. C.-à-d. que **-d:DProf** exécute le programme en utilisant le profileur Devel::DProf. Comme pour l'option `-M`, on peut passer des options au package Devel::foo et elles seront reçues et interprétées par la routine Devel::foo::import. La liste d'options séparées par des virgules doit être placée après le caractère =. Si le **t** est présent, cela indique au débogueur que les fils d'exécution (les threads) sont utilisés dans le code à déboguer. Voir *perldebug*.

-D*lettres*

-D*nombre*

détermine les drapeaux (flags) de débogage de Perl. Pour voir comment perl exécute votre programme, utilisez **-Dtls**. (Cela ne fonctionne que si le support du débogage est compilé dans votre interpréteur Perl). Une autre valeur intéressante est **-Dx**, qui affiche l'arbre syntaxique compilé. Et **-Dr** affiche les expressions rationnelles compilées ; le format de cette sortie est expliqué dans *perldebguts*.

Vous pouvez fournir un nombre à la place d'une listes de lettres (par exemple, **-D14** est équivalent à **-Dtls**):

```
       1  p  Découpage en unités lexicales et analyse
       2  s  Clichés de la pile (avec v, affiche toutes les piles)
       4  l  Traitement des piles de contextes de boucles
             (Context (loop) stack processing)
       8  t  Exécution tracée (Trace execution)
      16  o  Résolution et surcharge de méthodes
      32  c  Conversions chaînes/nombres
      64  P  Affiche les commandes du pré-processeur pour -P
     128  m  Allocation mémoire
     256  f  Traitement des formats
     512  r  Analyse et exécution des expressions rationnelles
    1024  x  Affichage de l'arbre syntaxique
    2048  u  Vérification des données souillées (tainted)
    4096     (Obsolète, utilisé auparavant pour les fuites mémoire)
    8192  H  Affiche les tables de hachage (usurps values())
   16384  X  Allocation scratchpad
   32768  D  Nettoyage
   65536  S  Synchronisation des fils d'exécution
  131072  T  Découpage en unités lexicales
  262144  R  Inclure le compteur de références des variables affichés
             (utilisé avec -Ds)
  524288  J  Ne pas s,t,P-déboguer (Sauter) les opcodes du package DB
 1048576  v  Verbosité: à utiliser avec d'autres drapeaux
 8388608  q  muet - pour l'instant, ne supprime que le message "EXECUTING"
```

Tous ces drapeaux nécessitent **-DDEBUGGING** quand vous compilez l'exécutable Perl (mais voyez tout de même *Devel::Peek* et *re* qui peuvent changer cela). Voir le fichier *INSTALL* dans la distribution des sources de Perl pour savoir comment le faire. Ce drapeau est automatiquement rajouté si vous compilez avec l'option `-g` quand `Configure` vous demande les drapeaux de votre optimiseur/débogueur.

Si vous essayez juste d'obtenir l'affichage de chaque ligne de code Perl au fur et à mesure de l'exécution, à la façon dont `sh -x` le fournit pour les scripts shell, vous ne pouvez pas utiliser l'option **-D** de Perl. Faites ceci à la place

```
# SI vous disposez de l'utilitaire "env"
env PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2" perl -dS program

# Syntaxe Bourne shell
$ PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2" perl -dS program

# Syntaxe csh
% (setenv PERLDB_OPTS "NonStop=1 AutoTrace=1 frame=2"; perl -dS program)
```

Voir *perldebug* pour plus de détails et des variantes.

-e *lignedeprogramme*

peut être utilisé pour entrer une ligne de programme. Si **-e** est présent, Perl ne cherchera pas de nom de fichier dans la liste des arguments. Plusieurs options **-e** peuvent être combinées pour créer des scripts multi-lignes. Assurez-vous de bien mettre les points-virgules là où vous en mettriez dans un programme normal.

-f

Désactive l'exécution de *$Config{sitelib}/sitecustomize.pl* au démarrage.

Perl peut être compilé de manière à exécuter, par défaut, le script *$Config{sitelib}/sitecustomize.pl* au démarrage. Ceci permet à l'administrateur de modifier le comportement de perl. Par exemple pour ajouter des répertoires au tableau @INC afin que perl trouve des modules à des emplacement non standard.

-F*motif*

indique le motif à utiliser pour l'auto-découpage si **-a** est aussi présent. Le motif peut être délimité par `//`, `""` ou `''`, sinon il sera mis entre apostrophes.

-h

affiche un résumé des options.

-i[*extension*]

indique que les fichiers traités par la forme <> doivent être édités sur place. Cela est accompli en renommant le fichier source, en ouvrant le fichier résultat sous le nom initial puis en sélectionnant ce fichier de résultat comme sortie par défaut pour les instructions print(). L'extension, si elle est fournie, est utilisée pour modifier le nom du fichier source pour faire une copie de sauvegarde, suivant ces règles :

Si aucune extension n'est fournie, aucune sauvegarde n'est faite et le fichier source est écrasé.

Si l'extension ne contient pas le caractère `*`, elle est ajoutée à la fin du nom de fichier courant comme suffixe. Si l'extension contient un ou plusieurs caractères `*`, chacune des `*` est remplacée par le nom de fichier courant. En perl, on pourrait l'écrire ainsi :

```
($backup = $extension) =~ s/\*/$file_name/g;
```

Cela vous permet d'ajouter un préfixe au fichier de sauvegarde, au lieu (ou en plus) d'un suffixe :

```
$ perl -pi'orig_*' -e 's/bar/baz/' fileA    # sauvegarde en 'orig_fileA'
```

Ou même de placer les sauvegardes des fichiers originaux dans un autre répertoire (à condition que ce répertoire existe déjà):

```
$ perl -pi'old/*.orig' -e 's/bar/baz/' fileA # sauvegarde en 'old/fileA.orig'
```

Ces exemples sont équivalents :

```
$ perl -pi -e 's/bar/baz/' fileA            # écrase le fichier courant
$ perl -pi '*' -e 's/bar/baz/' fileA        # écrase le fichier courant

$ perl -pi '.orig' -e 's/bar/baz/' fileA    # sauvegarde en 'fileA.orig'
$ perl -pi '*.orig' -e 's/bar/baz/' fileA   # sauvegarde en 'fileA.orig'
```

À partir du shell, écrire

```
$ perl -p -i.orig -e "s/foo/bar/; ... "
```

revient au même qu'utiliser le programme :

```
#!/usr/bin/perl -pi.orig
s/foo/bar/;
```

qui est équivalent à :

```
#!/usr/bin/perl
$extension = '.orig';
LINE: while (<>) {
    if ($ARGV ne $oldargv) {
        if ($extension !~ /\*/) {
            $backup = $ARGV . $extension;
        }
        else {
            ($backup = $extension) =~ s/\*/$ARGV/g;
        }
        rename($ARGV, $backup);
        open(ARGVOUT, ">$ARGV");
        select(ARGVOUT);
        $oldargv = $ARGV;
    }
    s/foo/bar/;
}
continue {
    print;  # affiche le nom du fichier original
}
select(STDOUT);
```

si ce n'est que l'option **-i** ne compare pas $ARGV et $oldargv pour savoir quand le nom de fichier a changé. Cette option utilise par contre, ARGVOUT pour l'identificateur de fichier (filehandle). Notez que STDOUT est restauré comme sortie par défaut après la boucle.

Comme montré ci-dessus, Perl crée le fichier de sauvegarde même si la sortie n'est pas modifiée. C'est donc une manière amusante de copier des fichiers.

```
$ perl -p -i '/some/file/path/*' -e 1 file1 file2 file3...
```

ou

```
$ perl -p -i '.orig' -e 1 file1 file2 file3...
```

Vous pouvez utiliser `eof` sans parenthèses pour localiser la fin de chaque fichier d'entrée, au cas où vous voulez ajouter des choses à la fin ou réinitialiser le comptage des lignes (cf. exemples dans eof in *perlfunc*).

Si, pour un fichier donné, Perl n'est pas capable de créer de fichier de sauvegarde avec l'extension indiquée, il ne traitera pas le fichier et passera au suivant (s'il existe).

Pour une discussion des détaillée des permissions des fichiers et **-i**, voir Pourquoi Perl me laisse effacer des fichiers protégés en écriture ? Pourquoi -i écrit dans des fichiers protégés ? N'est-ce pas un bug de Perl ? in *perlfaq5*

Vous ne pouvez pas utiliser **-i** pour créer des répertoires ou pour enlever des extensions à des fichiers.

Perl ne transforme pas les ~ dans les noms de fichiers, ce qui est une bonne chose, puisque certains l'utilisent pour leurs fichiers de sauvegarde :

```
$ perl -pi~ -e 's/foo/bar/' file1 file2 file3...
```

Remarquez que, puisque **-i** renomme ou efface le fichier original avant de créer un nouveau fichier du même nom, les liens hard d'Unix ne sont pas préservés.

Enfin, l'option **-i** n'empêche pas l'exécution si aucun fichier n'est fourni sur la ligne de commande. Dans ce cas, aucune sauvegarde n'est faite (puisque l'original ne peut être déterminé) et le traitement se fait de STDIN vers STDOUT, comme on peut s'y attendre.

-I*repertoire*

Les répertoires spécifiés pas **-I** sont ajoutés en tête du chemin de recherche des modules (`@INC`) et indiquent au pré-processeur C où chercher les fichiers à inclure. Le pré-processeur C est appelé avec **-P** ; par défaut il cherche dans /usr/include et /usr/lib/perl.

-l[*octval*]

permet le traitement automatique des fins de lignes. Cela a deux effets différents. Premièrement, `$/` (le séparateur d'enregistrements en entrée) est automatiquement enlevé (par chomp()) si utilisé en combinaison avec l'une des options **-n** ou **-p**. Deuxièmement, `$\` (le séparateur d'enregistrements en sortie) reçoit la valeur *octval* de telle manière que toutes les instructions print aient ce séparateur ajouté à la fin [du print()]. Si *octval* est omis, `$\` prend la valeur de `$/`. Par exemple, pour limiter les lignes à 80 colonnes :

```
perl -lpe 'substr($_, 80) = ""'
```

Notez que l'affectation `$\ = $/` est faite quand l'option est rencontrée, donc le séparateur en entrée peut être différent du séparateur en sortie si l'option **-l** est suivie de l'option **-0** :

```
gnufind / -print0 | perl -ln0e 'print "found $_" if -p'
```

Cela affecte un retour-chariot '\n' à `$\` et ensuite affecte le caractère nul (ASCII 0) à `$/`.

-m[-]*module*

-M[-]*module*

-M[-]*'module ...'*

-[mM][-]*module=arg[,arg]...*

-m*module* exécute `use` *module* `();` avant d'exécuter votre programme.

-M*module* exécute `use` *module* `;` avant d'exécuter votre programme. Vous pouvez utiliser des apostrophes pour ajouter du code supplémentaire après le nom du module, par exemple, `'-Mmodule qw(toto titi)'`.

Si le premier caractère après **-M** ou **-m** est un moins (-), alors le 'use' est remplacé par 'no'.

Un peu de sucre syntaxique intégré permet d'écrire **-mmodule=toto,titi** ou **-Mmodule=toto,titi** comme un raccouci de `'-Mmodule qw(toto titi)'`. Cela évite de recourir à des apostrophes lorsqu'on importe des symboles. Le code généré par **-Mmodule=toto,titi** est `use module split(/,/,q{toto,titi})`. Notez que la forme = fait disparaître la distinction entre **-m** et **-M**.

Par conséquent **-MFoo=number** n'effectuera jamais une vérification de version (à moins que la routine `Foo::import()` la fasse elle-même ce qui peut arriver si Foo hérite de Exporter par exemple).

-n

indique à Perl de considérer votre programme comme étant entouré par la boucle suivante, qui le fait itérer sur les noms de fichiers passés en arguments, à la manière de **sed -n** ou **awk** :

```
LINE:
  while (<>) {
      ...             # votre script va là
  }
```

Notez que les lignes ne sont pas affichées par défaut. Cf. **-p** pour afficher les lignes traitées. Si un fichier passé en argument ne peut pas être ouvert, pour quelque raison que ce soit, Perl vous prévient, et passe au fichier suivant.

Voici un moyen efficace d'effacer tous les fichiers agés de plus d'une semaine :

```
find . -mtime +7 -print | perl -nle unlink
```

C'est plus rapide que d'utiliser l'option **-exec** de **find**, car vous ne lancez plus un processus pour chaque fichier à effacer. Cela souffre du bug entraînant une mauvaise gestion des fins de lignes dans les chemins, que vous pouvez régler si vous suivez l'exemple donné pour l'option **-0**.

Les blocs `BEGIN` et `END` peuvent être utilisés pour prendre le contrôle des opérations, avant ou après la boucle implicite du programme, comme dans **awk**.

-p

indique à Perl de considérer votre programme comme étant entouré par la boucle suivante, qui le fait itérer sur les noms de fichiers passés en arguments, un peu à la manière de **sed** :

```
LINE:
  while (<>) {
     ...             # votre programme va là
  } continue {
     print or die "-p destination : $!\n";
  }
```

Si un fichier passé en argument ne peut pas être ouvert, pour quelque raison que ce soit, Perl vous prévient, et passe au fichier suivant. Notez que les lignes sont affichées automatiquement. Une erreur durant l'affichage est considérée comme fatale. Pour supprimmer les affichages, utilisez l'option **-n**. L'option **-p** prend le dessus sur **-n**.

Les blocs `BEGIN` et `END` peuvent être utilisés pour prendre le contrôle des opérations, avant ou après la boucle implicite, comme dans **awk**.

-P <-P>

NOTE : l'usage de -P est très fortement déconseillé à cause des problèmes que cela pose, et en particulier une mauvaise portabilité.

demande à Perl de faire passer votre programme dans le pré-processeur C avant la compilation. Étant donné que les commentaires Perl comme les directives de **cpp** commencent par le caractère #, il vaut mieux éviter de mettre des commentaires qui débutent par un mot réservé par le pré-processeur tels que `"if"`, `"else"` ou `"define"`.

Si vous envisagez d'utiliser `-P`, vous devriez aussi jeter un oeil au module Filter::cpp sur CPAN.

Voici une liste non-exhaustive des problèmes inhérents à -P :

- La ligne `#!` est supprimée, donc aucune option n'est prise en compte.
- Un `-P` dans la ligne `#!` ne fonctionne pas.
- **Toutes** les lignes qui commencent par un `#` (précédé éventuellement d'espaces) mais qui ne ressemblent pas à une commande cpp sont supprimées y compris celles dans les chaînes Perl, dans les expression rationnelles et dans les here-docs.
- Sur certaines platformes le pré-processeur C en sait trop : il reconnait les commentaires à la C++, ceux qui commencent par `"//"` et qui se terminent en fin de ligne. En conséquence, cela pose problème pour des constructions Perl courantes telle :

  ```
  s/foo//;
  ```

  qui, après -P, devienne du code illégal :

  ```
  s/foo
  ```

  Un moyen de contourner cela consiste à utiliser un autre séparateur que `"/"` comme, par exmeple, `"!"` :

  ```
  s!foo!!;
  ```
- Outre un pré-processeur C fonctionnel, cela nécessite aussi un *sed* fonctionnel. Si vous n'êtes pas sur UNIX, vous risquez de ne pas avoir cette chance.
- Le numéro des lignes du script ne sont pas préservés.
- L'option `-x` ne fonctionne pas avec `-P`.

-s

active une analyse rudimentaire des arguments sur la ligne de commande situés après le nom du programme mais avant tout nom de fichier passé en argument (ou avant un **–**). Toute option trouvée là est retirée de @ARGV et définit la variable éponyme dans le programme Perl. Le programme suivant affiche "1" si et seulement si le programme est invoqué avec une option **-xyz**, et "abc" s'il est invoqué avec **-xyz=abc**.

```
#!/usr/bin/perl -s
if ($xyz) { print "$xyz\n"; }
```

Notez qu'un argument comme **–help** créra la variable ${-help} qui n'est pas conforme avec `strict refs`. De plus, si vous utilisez cette option dans un script qui active les avertissements (use warnings), vous obtiendrez de nombreux avertissements "used only once" ("utilisé une seule fois").

-S

fait en sorte que Perl utilise la variable d'environnement PATH pour rechercher le programme (sauf si le nom du programme contient des séparateurs de répertoires).

Sur certaines plateformes, Perl ajoute des suffixes au nom du programme pendant sa recherche. Par exemple, sur les plateformes Win32, les suffixes ".bat" et ".cmd" sont ajoutés si la recherche du nom originel échoue et si le nom ne se termine pas déjà par l'un de ces suffixes. Si votre Perl a été compilé avec DEBUGGING activé, donner l'option -Dp à Perl montre la progression de la recherche.

Ceci est typiquement utilisé pour émuler le démarrage #! sur les plateformes qui ne le supportent pas. C'est aussi pratique pour déboguer un script qui utilise #! et qui doit donc être normalement trouvé par le mécanisme de recherche via PATH du shell.

Cet exemple fonctionne sur de nombreuses plateformes ayant un shell compatible avec le Bourne shell :

```
#!/usr/bin/perl
eval 'exec /usr/bin/perl -wS $0 ${1+"$@"}'
        if $running_under_some_shell;
```

Le système ignore la première ligne et donne le programme à */bin/sh*, qui essaye alors d'exécuter le programme Perl comme un script shell. Le shell exécute la deuxième ligne comme une commande normale, et ainsi démarre l'interpréteur Perl. Sur certains systèmes, $0 ne contient pas toujours le chemin complet, l'option **-S** dit donc à Perl de rechercher le programme si nécessaire. Après que Perl ait localisé le programme, il analyse les lignes et les ignore car la variable $running_under_some_shell n'est jamais vraie. Si le programme doit être interprété par csh, vous devrez remplacer `${1+"$@"}` par `$*`, même si cela ne comprend pas les espaces (et les autres caractères équivalents) inclus dans la liste d'arguments. Pour démarrer sh plutôt que csh, certains systèmes peuvent nécessiter le remplacement de la ligne #! par une contenant juste un deux points, qui sera poliment ignoré par Perl. D'autres

systèmes ne peuvent pas contrôler cela, et ont besoin d'une construction totalement diabolique qui fonctionnera sous **csh**, **sh**, ou Perl, comme celle-ci :

```
eval '(exit $?0)' && eval 'exec perl -wS $0 ${1+"$@"}'
& eval 'exec /usr/bin/perl -wS $0 $argv:q'
        if $running_under_some_shell;
```

Si le nom de fichier fourni contient des séparateurs de répertoires (i.e. si c'est un chemin absolu ou relatif), et si le fichier n'est pas trouvé, les plateformes qui ajoutent des extensions de noms de fichiers le feront et essayeront de trouver le fichier avec ces extensions ajoutées, les unes après les autres.

Sur les plateformes compatibles DOS, si le programme ne contient pas de séparateurs de répertoires, il sera d'abord recherché dans le répertoire courant avant d'être recherché dans le PATH. Sur les plateformes Unix, le programme sera recherché strictement dans le PATH.

-t

Agit comme **-T** mais les vérifications de "souillure" n'engendrent que des avertissements au lieu d'une erreur fatale. Ces avertissements peuvent même être désactivés via `no warnings qw(taint)`.

NOTE : ce n'est qu'un substitut de -T qui ne devrait être utilisé que temporairement pour aider à la sécurisation d'un ancien code. Pour du code réellement utilisé en production ou pour du nouveau code écrit à partir de la feuille blanche, vous devriez toujours utiliser **-T**.

-T

force l'activation des vérifications de "souillure" (des données) pour que vous puissiez les tester. Ordinairement, ces vérifications sont faites seulement lorsqu'on exécute en setuid ou setgid. C'est une bonne idée de les activer explicitement pour les programmes exécutés à la demande de quelqu'un en qui vous n'avez pas une totale confiance, comme par exemple les programmes CGI ou tout serveur internet que vous pourriez écrire en Perl. Voir *perlsec* pour plus de détails. Pour des raisons de sécurité, cette option doit être vue par Perl très tôt ; cela signifie habituellement qu'elle doit apparaître le plus tôt possible dans la ligne de commande ou sur la ligne #! pour les systèmes qui le supportent.

-u

Cette option obsolète force Perl à écrire un coredump (une image mémoire) après la compilation de votre programme. Vous pouvez ensuite théoriquement prendre ce coredump et en faire un fichier exécutable en utilisant le programme **undump** (non fourni). Ceci accélère le démarrage au détriment de l'espace disque (que vous pouvez minimiser en strippant l'exécutable. Mais un exécutable "hello world" fait encore environ 200K sur ma machine). Si vous voulez exécuter une portion de votre programme avant le dump, utilisez l'opérateur dump() à la place. Note : la disponibilité de **undump** est spécifique à la plateforme et il peut donc ne pas être disponible pour un portage spécifique de Perl.

Cette option a été remplacée par le nouveau générateur de code du compilateur Perl. Voir *B* et *B::Bytecode* pour plus de détails.

-U

permet à Perl de réaliser des opérations non sûres. Actuellement, les seules opérations "non sûres" sont l'effacement des répertoires par unlink lors d'une exécution en tant que superutilisateur, et l'exécution de programmes setuid avec les vérifications de souillures fatales transformées en avertissements. Notez que l'option **-w** (ou la variable `$^W`) doit être utilisée en même temps que cette option pour effectivement *générer* les avertissements de vérification de souillure.

-v

affiche le numéro de version et le niveau de mise à jour (le niveau de patch) de votre exécutable perl.

-V

affiche un résumé des principales valeurs de configuration de perl et le contenu actuel de @INC.

-V:*nom*

affiche sur STDOUT la valeur de la variable de configuration dont le nom est fourni. Par exemple :

```
$ perl -V:libc
    libc='/lib/libc-2.2.4.so';
$ perl -V:lib.
    libs='-lnsl -lgdbm -ldb -ldl -lm -lcrypt -lutil -lc';
    libc='/lib/libc-2.2.4.so';
$ perl -V:lib.*
    libpth='/usr/local/lib /lib /usr/lib';
    libs='-lnsl -lgdbm -ldb -ldl -lm -lcrypt -lutil -lc';
    lib_ext='.a';
    libc='/lib/libc-2.2.4.so';
```

```
libperl='libperl.a';
....
```

De plus, des deux-points supplémentaires permettent de contrôler le format d'affichage. Un deux-points final supprime les passages à la ligne et les ';' finaux, autorisant ainsi l'utilisation de cette option dans une ligne de commande du shell. (Moyen mnémotechnique : le séparateur de chemin ':'.)

```
$ echo "compression-vars: " `perl -V:z.*: ` " are here !"
compression-vars:  zcat='' zip='zip'  are here !
```

Un deux-points supplémentaire en tête supprime le 'name=' de la réponse, autorisant ainsi l'ajout du nom que vous voulez. (Moyen mnémotechnique : étiquette vide.)

```
$ echo "goodvfork="`./perl -Ilib -V::usevfork`
goodvfork=false;
```

Les deux-points en tête et final peuvent être utilisés conjointement si vous voulez des valeurs sans nom. Remarques que dans le cas ci-dessous, les valeurs des variables de configuration de PERL_API sont fournies dans l'ordre alphabétique (de leur nom).

```
$ echo building_on `perl -V::osname: -V::PERL_API_.*:` now
building_on 'linux' '5' '1' '9' now
```

-w

affiche des avertissements concernant les constructions douteuses, telles que les noms de variables mentionnés une seule fois, les variables scalaires utilisées avant d'être définies, les sous-programmes redéfinis, les références aux handles de fichiers indéfinis ou aux handles de fichiers ouverts en lecture seule et sur lesquels vous essayez d'écrire, les valeurs utilisées comme des nombres et qui n'ont pas l'air d'être des nombres, les tableaux utilisés comme s'ils étaient des scalaires, les sous-programmes récursifs se rappellant eux-mêmes plus de 100 fois et d'innombrables autres choses.

Cette option ne fait que valider la variable interne `^$W`. Vous pouvez invalider certains avertissements ou les transformer en erreurs fatales de façon spécifique en cpatant les signaux `__WARN__`, comme décrit dans *perlvar* et dans warn in *perlfunc*. Voir aussi *perldiag* et *perltrap*. Une nouvelle façon de gérer finement les avertissements est aussi disponible si vous voulez en manipuler des classes entières ; voir *warnings* ou *perllexwarn*.

-W

valide tous les avertissements sans tenir compte de `no warnings` ou de `$^W`. Voir *perllexwarn*.

-X

invalide tous les avertissements sans tenir compte de `use warnings` ou de `$^W`. Voir *perllexwarn*.

-x

-x *répertoire*

indique à Perl que le programme est contenu dans un grand texte ACSII n'ayant rien à voir, comme par exemple un courrier électronique. Les déchets qui le précèdent seront ignorés jusqu'à la première ligne commençant par #! et contenant la chaîne "perl". Toute option significative sur cette ligne sera appliquée. Si un nom de répertoire est spécifié, Perl passera dans ce répertoire avant d'exécuter le programme. L'option **-x** contrôle seulement la destruction des déchets qui précèdent le script. Le programme doit se terminer par `__END__` si des déchets doivent être ignorés après lui (si on le désire, le programme peut traiter tout ou partie de ce qui le suit via le handle de fichier DATA).

4 ENVIRONNEMENT

HOME

Utilisée si chdir n'a pas d'argument.

LOGDIR

Utilisée si chdir n'a pas d'argument et si HOME n'est pas définie.

PATH

Utilisée lors de l'exécution de sous-processus, et dans la recherche du programme si **-S** est utilisée.

PERL5LIB

Une liste de répertoires dans lesquels on doit rechercher les fichiers de bibliothèques Perl avant de les rechercher dans la bibliothèque standard et le répertoire courant. Tous les sous-répertoires spécifiques à l'architecture courante sont automatiquement inclus s'ils existent aux endroits spécifiés. Si PERL5LIB n'est pas définie, PERLLIB est utilisée. Les noms des répertoires sont séparés (comme pour PATH) par des deux-points sur les plateformes de type

Unix et par des points-virgules sur Windows (le bon séparateur de répertoires est donné par la commande `perl -V:path_sep`).

Lors d'une exécution avec vérifications de souillure activée (soit parce que le programme est exécuté en setuid ou en setgid, soit parce que l'option **-T** est activée), aucune de ces deux variables n'est utilisée. Le script devrait dire à la place :

```
use lib "/my/directory";
```

PERL5OPT

Options de ligne de commande. Les options dans cette variable sont considérées comme faisant partie de toute ligne de commande Perl. Seules les options **-[DIMUdmtw]** sont autorisées. Lors d'une exécution avec vérifications de souillure (si le programme est exécuté en setuid ou si setgid, ou si l'option **-T** est activée), cette variable est ignorée. Si PERL5OPT commence par **-T**, la vérification sera activée et toute les options qui suivent seront ignorées.

PERLIO

Une liste de filtres PerlIO utilisant l'espace ou le deux-points comme séparateur. Si perl est construit afin d'utiliser le système d'E/S PerlIO (c'est le cas par défaut) alors ces filtres sont utilisés pour les E/S de perl.

Par convention, on commence une suite de noms de filtres par deux-points (par exemple `:perlio`) pour mettre en évidence la similarité avec des "attributs" variables. Mais le code qui analyse cette chaîne de spécifications de filtres (et qui décode aussi la variable d'environnement PERLIO) considère les deux-points comme un séparateur.

Une variable PERLIO non définie ou vide est équivalent à `:stdio`.

Cette liste devient la valeur par défaut pour *toutes* les E/S de perl. Par conséquent, seuls les filtres déjà intégrés à perl peuvent apparaître dans cette liste. Les filtres externes (comme par exemple :encoding()) nécessitent des E/S utilisables pour pouvoir être chargés ! Voir "directive open" pour savoir comment ajouter des codages externes aux valeurs par défaut.

Voici une présentation rapide des filtres utilisables dans la variable d'encoding PERLIO. Pour plus de détails, consultez *PerlIO*.

:bytes

Un pseudo-filtre qui désactive le drapeau `:utf8` pour le filre précédent. Rarement utilisable seul dans la variable d'environnement PERLIO. Peut-être intéressant dans des cas comme `:crlf:bytes` ou `:perlio:bytes`.

:crlf

Un filtre qui assure la traduction entre CRLF et "\n" en distinguant les fichiers "texte" et les fichiers "binaires" à la manière de MS-DOS et des systèmes d'exploitation assimilés. (Pour l'instant, il ne pousse pas le mimétisme jusqu'à considérer un Control-Z comme un marqueur de fin de fichier.)

:mmap

Un filtre qui implémente la "lecture" des fichiers via `mmap()` pour rendre accessible l'ensemble du fichier dans l'espace mémoire adressable du processus et pour l'utiliser comme "tampon" (buffer) PerlIO.

:perlio

C'est une ré-écriture sous forme de filtre PerlIO des méthodes de gestion des buffers à la "stdio". Pour toutes es opérations, il fait donc appel aux couches sous-jacentes (classiquement `:unix`).

:pop

Un pseudo-filtre expérimental qui permet de retirer le filtre le plus haut. À utiliser avec des pincettes, comme si c'était de la nitroglycérine.

:raw

Un pseudo-filtre qui manipule les autres filtres. L'application du filtre `:raw` est équivalent à un appel à `binmode($fh)`. Chaque octet du flux est passé tel quel sans aucun traduction. En particulier la traduction des CRLF ou les filtres :utf8 déduits du locale courant sont désactivés.

Contrairement aux versions antérieures de Perl, `:raw` n'est *plus* l'exact inverse de `:crlf` - les autres filtres qui pourraient modifier la nature binaire du flix sont aussi supprimés ou désactivés.

:stdio

Ce filtre fournit une interface PerlIO pour accéder aux appels ANSI C de la couche "stdio" du système. Ce filtre fournit à la fois le tamponnage (la bufferisation) et les E/S. Notez que le filtre `:stdio` n'inclut pas la traduction du CRLF même si c'est le comportement normal sur la plateforme. Vous devez ajouter un filtre `:crlf` au-dessus pour le faire.

:unix

Filtre de bas niveau qui fait appel à `read`, `write`, `lseek`, etc.

:utf8

Un pseudo-filtre qui indique aux autres filtres que la sortie doit être en utf8 ou que les entrées doivent être considérées comme étant déjà sous la forme utf8. Peut-être utile dans la variable d'environnement PERLIO pour placer l'UTF-8 comme valeur par défaut. (Pour désactiver cela, utilisez le filtre `:bytes`.)

:win32

Sur les plateformes Win32, ce filtre *expérimental* utilise les manipulateur d'E/S natif plutôt que les descripteurs numériques de fichier à la unix. Est reconnu comme étant bogué dans cette version.

Sur chacune des plateformes, l'ensemble de filtres choisis par défaut devrait donner des résultats corrects.

Pour les plateformes Unix, cela signifie "unix perlio" ou "stdio". Configure est réglé pour préférer l'implémentation "stdio" si la bibliothèque système propose des accès rapides aux tampons. Sinon, c'est "unix perlio" qui est proposé.

Sur Win32, le réglage par défaut de la distribution actuelle est "unix crlf". Le "stdio" de Win32 utilisé comme IO pour perl a trop de bugs ou de fonctionnalités bancales qui, de plus, dépendent parfois du fournisseur ou de la version du compilateur C utilisé. En utilisant notre propre filtre `crlf` pour la bufferisation permet d'éviter ces problèmes et rend les chose plus uniformes. Le filtre `crlf` fournit la conversion CRLF en "\n" et inversement, mais aussi les fonctionnalités de bufferisation (mise en mémoire tampon).

La distribution actuelle utilise `unix` comme second filtre sur Win32. Elle utilise donc les routines à base de descripteurs numériques de fichiers du compilateur C. C'est une couche native expérimentale pour `win32` qui devrait s'améliorer et deviendra un jour le choix par défaut pour Win32.

PERLIO_DEBUG

Si cette variable contient le nom d'un fichier ou d'un périphérique alors certaines opérations du sous-système PerlIO seront tracées sur ce fichier (ouvert en ajout). Une utilisation classique sur Unix :

```
PERLIO_DEBUG=/dev/tty perl script ...
```

et sur Win32 :

```
set PERLIO_DEBUG=CON
perl script ...
```

Cette fonctionnalité n'est pas utilisable pour les script setuid ou ceux qui utilisent l'option **-T**.

PERLLIB

Une liste de répertoires ponctuée de deux points dans lesquels on doit rechercher les fichiers de bibliothèque de Perl avant de les rechercher dans la bibliothèque standard et le répertoire courant. Si PERL5LIB est définie, PERLLIB n'est pas utilisée.

PERL5DB

La commande utilisée pour charger le code du débogueur. La valeur par défaut est :

```
BEGIN { require 'perl5db.pl' }
```

PERL5DB_THREADED

Si cette variable contient une valeur vraie, cela indique au débogueur que le code à déboguer utilise les fils d'exécution (les threads).

PERL5SHELL (spécifique à la version WIN32)

Peut être fixée vers un shell alternatif que perl doit utiliser en interne pour exécuter les commandes fournie entre accents graves ou par system(). La valeur par défaut est `cmd.exe /x/d/c` sous WindowsNT et `command.com /c` sous Windows95. La valeur est considérée comme délimitée par des espaces. Précédez tout caractère devant être protégé (comme un espace ou une barre oblique inverse) par une barre oblique inverse.

Notez que Perl n'utilise pas COMSPEC pour cela car COMSPEC a un haut degré de variabilité entre les utilisateurs, ce qui amène à des soucis de portabilité. De plus, perl peut utiliser un shell qui ne convienne pas à un usage interactif, et le fait de fixer COMSPEC vers un tel shell peut interférer avec le fonctionnement correct d'autres programmes (qui regardent habituellement COMSPEC pour trouver un shell permettant l'utilisation interactive).

PERL_ALLOW_NON_IFS_LSP (spécifique à la version Win32)

Si cette variable vaut 1, cela autorise l'utilisation de LSP non compatibles ISF. Perl essaye normalement d'utiliser des LSP compatibles ISF afin de permettre l'émulation de véritables filehandles au-dessus des sockets de Windows. Mais cela peut poser problème si vous utilisez un firewall tel que McAfee Guardian qui exige que toutes les applications utilisent ses propres LSP qui ne sont pas compatibles ISF. Perl ne le fera pas évidemment. En plaçant cette variable à 1, Perl utilisera le premier LSP utilisable dans le catalogue ce qui contentera McAfee Guardian (et, dans ce cas particulier, Perl continuera à fonctionner puisque les LSP de McAfee Guardian permettent, par des moyens détournés, le fonctionnement des applications exigeant la compatibilité IFS).

PERL_DEBUG_MSTATS

Valable uniquement si perl est compilé avec la fonction malloc inclue dans la distribution de perl (c.-à-d. si `perl -V:d_mymalloc` vaut 'define'). Si elle a une valeur, cela provoque l'affichage de statistiques d'utilisation de la mémoire après l'exécution. Si sa valeur est un entier supérieur à un, les statistiques d'utilisation de la mémoire sont aussi affichées après la phase de compilation.

PERL_DESTRUCT_LEVEL

Valable uniquement si votre exécutable perl a été construit avec **-DDEBUGGING**, ceci contrôle le comportement de la destruction globale des objets et autres références. Voir PERL_DESTRUCT_LEVEL in *perlhack* pour plus d'informations.

PERL_DL_NONLAZY

Si cette variable vaut 1, perl tente de résoudre **tous** les symboles non-définis lors du chargement d'une bibliothèque dynamique. Le comportement par défaut est d'attendre l'utilisation d'un symbole pour effectuer sa résolution. En activant cette variable durant les tests d'une extension, vous détecterez les noms de fonctions mal orthographiés même si les test n'y font pas appel.

PERL_ENCODING

Si vous utilisez la directive `encoding` sans fournir explicitement le nom du codage, la variable d'environnement PERL_ENCODING est utilisé comme nom de codage.

PERL_HASH_SEED

(Existe depuis Perl 5.8.1.) Sa valeur est utilisée pour contrôler le caractère aléatoire de le fonction de hachage de Perl. Pour simuler le comportement des versions antérieures à la version 5.8.1, placez-y une valeur entière (avec la valeur zéro vous retrouverez le comportement exact de la version 5.8.0). Dans les versions antérieures à la 5.8.1, entre autres, l'ordre des clés des tables de hachage étaient toujours le même d'une exécution à l'autre de Perl.

Le comportement par défaut est de choisir un valeur aléatoire à moins que la variable PERL_HASH_SEED soit définie. Si Perl a été compilé avec l'option `-DUSE_HASH_SEED_EXPLICIT`, le comportement par défaut et de **ne pas** choisir une valeur aléatoire à moins que PERL_HASH_SEED soit définie.

Si PERL_HASH_SEED n'est pas définie ou ne contient pas une valeur numérique, Perl utilise une graine pseudo-aléatoire fournie par le système et ses bibliothèques. Cela signifie que chaque exécution de Perl proposera un ordre différent pour keys(), values() et each().

Notez bien que la valeur de cette graine est une information sensible. L'ordre dans les tables de hachage est modifié aléatoirement pour protéger le code Perl d'attaque locale ou distante. En définissant manuellement cette graine, cette protection peut être partiellement ou complètement perdue.

Voir Attaques par complexité algorithmique in *perlsec* et PERL_HASH_SEED_DEBUG pour plus d'information.

PERL_HASH_SEED_DEBUG

(Existe depuis Perl 5.8.1) Placer cette variable d'environnement à un pour afficher (sur SDTERR) la valeur de la graine des tables de hachage au tout début de l'exécution. C'est prévu, en combinaison avec PERL_HASH_SEED pour aider au débogage malgré le comportement non-déterministe dû au caractère aléatoire de la fonction de hachage de Perl.

Notez bien que la valeur de cette graine est une information sensible. Sa connaissance permet de construire une attaque en déni de service contre Perl, même à distance. Voir Attaques par complexité algorithmique in *perlsec* et PERL_HASH_SEED_DEBUG pour plus d'information. **Ne divulguez donc pas la valeur de cette graine** à ceux qui n'ont pas à la connaître. Voir aussi hash_seed() dans *Hash::Util*.

PERL_ROOT (spécifique au portage VMS)

Un nom logique qui, sur VMS uniquement, traduit la racine cachée contenant perl et les chemins pour @INC. Parmi les autres noms logiques qui affectent perl sur VMS, on trouve PERLSHR, PERL_ENV_TABLES et SYS$TIMEZONE_DIFFERENTIAL mais ils sont optionnels et vous en saurez plus en lisant *perlvms* et le fichier *README.vms* dans la distribution des sources de Perl.

PERL_SIGNALS

Dans Perl 5.8.1 et au-delà. Si cette variable contient `unsafe` (NdT : *non-sûr*), le comportement des versions pré-Perl-5.8.0 vis-à-vis de signaux est adopté (délivrance immédiate mais non sûre). Si cette variable contient `safe` (NdT : *sûr*), le comportement sûr (et différé) est utilisé. Voir Signaux différés (signaux sûrs) in *perlipc*.

PERL_UNICODE

Équivalent à l'option **-C** de la ligne de commande. Notez que ce n'est pas une variable booléenne : y mettre la valeur `"1"` n'est pas le bon moyen pour "activer Unicode" (quelqu'en soit la signification). Par contre, vous pouvez "désactiver Unicode" en y plaçant la valeur `"0"` (ou en rendant PERL_UNICODE indéfini dans votre shell avant de démarrer Perl). Voir la description de l'option `-C` pour plus d'information.

Perl utilise aussi de variables d'environnement pour contrôler la manière dont il manipule les données spécifiques à un langage naturel particulier. Voir *perllocale*.

À part toutes celles-ci, Perl n'utilise pas d'autres variables d'environnement, sauf pour les rendre accessibles au programme en cours d'exécution et à ses processus fils. Toutefois, les programmes exécutés en setuid feraient bien d'exécuter les lignes suivantes avant de faire quoi que ce soit d'autres, ne serait-ce que pour que les gens restent honnêtes :

```
$ENV{PATH}  = '/bin:/usr/bin';    # ou ce que vous voulez
$ENV{SHELL} = '/bin/sh' if exists $ENV{SHELL};
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
```

5 TRADUCTION

5.1 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/.

5.2 Traducteur

Pour la version 5.6.0 : Loic Tortay <*loict@bougon.net*>, Roland Trique <*roland.trique@uhb.fr*>. Pour la mise à jour en 5.8.8 : Paul Gaborit (paul.gaborit @ enstimac.fr).

5.3 Relecture

Personne pour l'instant.

6 À propos de ce document

Ce document est la traduction française du document orginal distribué avec perl. Vous pouvez retrouver l'ensemble de la documentation française Perl (éventuellement mise à jour) en consultant l'URL <http://perl.enstimac.fr/>.

Ce document PDF a été 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 électronique dans la rubrique TRADUCTION) et expliquez-lui gentiment vos remarques ou critiques. Il devrait normalement vous répondre et prendre en compte votre avis. En l'absence de réponse, vous pouvez éventuellement me contacter.

Vous pouvez aussi participer à l'effort de traduction de la documentation Perl. Toutes les bonnes volontés sont les bienvenues. Vous devriez trouver tous les renseignements nécessaires en consultant l'URL ci-dessus.