

# perltrap

## Table des matières

<b>1</b>	<b>NAME/NOM</b>	<b>1</b>
<b>2</b>	<b>DESCRIPTION</b>	<b>1</b>
2.1	Pièges de awk	2
2.2	Pièges de C	2
2.3	Pièges de sed	3
2.4	Pièges du shell	3
2.5	Pièges de Perl	3
2.6	Pièges entre Perl4 et Perl5	4
2.7	Pièges liés aux corrections de bugs, aux désapprobations et aux abandons	4
2.8	Pièges de l'analyse syntaxique	7
2.9	Pièges numériques	7
2.10	Pièges des types de données généraux	8
2.11	Pièges du contexte - contextes scalaires et de liste	10
2.12	Pièges de la précedence (les priorités)	11
2.13	Pièges des expressions rationnelles générales lors de l'utilisation de s///, etc.	12
2.14	Pièges des sous-programmes, des signaux et des tris	13
2.15	Pièges du système d'exploitation	14
2.16	Pièges de l'interpolation	14
2.17	Pièges DBM	16
2.18	Pièges non classés	16
<b>3</b>	<b>TRADUCTION</b>	<b>17</b>
3.1	Version	17
3.2	Traducteur	17
3.3	Relecture	17
<b>4</b>	<b>À propos de ce document</b>	<b>17</b>

## 1 NAME/NOM

perltrap - Les pièges de Perl pour l'imprudent

## 2 DESCRIPTION

Le plus gros piège est d'oublier d'utiliser la directive `use warnings` ou l'option `-w` ; voir le manuel *perllexwarn* et le manuel *perlrun*. Le deuxième plus gros piège est de ne pas rendre la totalité de votre programme exécutable sous `use strict`. Le troisième plus grand piège est de ne pas lire la liste des changements effectués dans cette version de Perl ; voir le manuel *perldelta*.

## 2.1 Pièges de `awk`

Les utilisateurs avertis de `awk` devraient se soucier particulièrement de ce qui suit :

- Un programme Perl est exécuté une seule fois et non pour chacune des lignes d'entrée. Vous pouvez obtenir une boucle implicite en utilisant les options `-n` et `-p`.
- Le module `English`, chargé par
 

```
use English;
```

 vous permet de vous référer aux variables spéciales (comme `$/`) par des noms (comme `$RS`), comme si elles étaient en `awk` ; voir le manuel *perlvar* pour plus de détails.
- Le point-virgule est requis après toute instruction simple en Perl (sauf à la fin d'un bloc). La fin de ligne n'est pas un délimiteur d'instruction.
- Les accolades sont requises pour les `ifs` et les `whiles`.
- Les variables commencent par "\$", "@" ou "%" en Perl.
- Les tableaux sont indexés à partir de 0. De la même manière, les positions renvoyées par `substr()` et `index()` dans les chaînes sont comptées à partir de 0.
- Vous devez décider si votre tableau a pour indices des nombres ou des chaînes.
- Les hachages ne sont pas créés par une simple référence.
- Vous devez décider si une comparaison porte sur des chaînes ou des nombres.
- La lecture d'une entrée ne la découpe pas pour vous. Vous devez la transformer en un tableau vous-même. Et l'opérateur `split()` a des arguments différents qu'en `awk`.
- La ligne courante en entrée est normalement dans `$_`, pas dans `$0`. Elle n'a en général pas encore perdu sa fin de ligne (`$0` est le nom du programme exécuté). Voir le manuel *perlvar*.
- `$digit` ne se réfère pas à des champs – il se réfère aux sous-chaînes détectées par le dernier motif de correspondance.
- L'instruction `print()` n'ajoute pas de séparateurs de champs et d'enregistrements à moins que vous ne fixiez `$,` et `$/`. Vous pouvez fixer `$OFS` et `$ORS` si vous utilisez le module `English`.
- Vous devez ouvrir vos fichiers avant d'y écrire.
- L'opérateur d'intervalle est `..`, et pas la virgule. L'opérateur virgule fonctionne comme en C.
- L'opérateur de correspondance est `=~`, pas `~` ("`~`" est l'opérateur de complément à 1, comme en C).
- L'opérateur d'exponentiation est `**`, pas `^`. `^` est l'opérateur XOR, comme en C (vous savez, on pourrait finir par avoir la sensation que `awk` est intrinsèquement incompatible avec C).
- L'opérateur de concaténation est `.`, pas la chaîne vide (utiliser la chaîne vide rendrait `/pat/ /pat/` impossible à analyser lexicalement, car le troisième signe de division serait interprété comme un opérateur de division – le détecteur de mots-clés est en fait légèrement sensible au contexte pour les opérateurs tels que `/`, `?`, et `>`. Et en fait, `.` lui-même peut être le début d'un nombre).
- Les mots-clés `next`, `exit`, et `continue` fonctionnent différemment.
- Les variables suivantes fonctionnent différemment :

Awk	Perl
ARGC	scalar @ARGV (à comparer avec \$#ARGV)
ARGV[0]	\$0
FILENAME	\$ARGV
FNR	\$. - quelque chose
FS	(ce que vous voulez)
NF	\$#Fld, ou quelque chose comme ça
NR	\$.
OFMT	\$#
OFS	\$,
ORS	\$\ \$
RLENGTH	length(\$&)
RS	\$/
RSTART	length(\$`)
SUBSEP	\$;

Vous ne pouvez pas mettre un motif dans `$RS`, seulement une chaîne.

- En cas de doute, faites passer la construction `awk` dans `a2p` et voyez ce que cela donne.

## 2.2 Pièges de C

Les programmeurs C et C++ devraient prendre note de ce qui suit :

- Les accolades sont requises pour les `ifs` et les `whiles`.
- Vous devez utiliser `elsif` à la place de `else if`.

- Les mots-clés `break` et `continue` de C deviennent respectivement `last` et `next` en Perl. Contrairement au C, ceux-ci ne fonctionnent *pas* à l'intérieur d'une structure `do { } while`. Voir le titre *Contrôle de Boucle* dans le manuel *perlsyn*.
- Il n'y a pas d'instruction `switch` (mais il est aisé d'en construire une à la volée, voir le titre *BLOCs de Base et Instruction Switch* dans le manuel *perlsyn*).
- Les variables commencent par "\$", "@" ou "%" en Perl.
- Les commentaires commencent par "#", pas par "/\*" ou "/\*". Perl peut interpréter un commentaire C/C++ comme un opérateur de division, une expression rationnelle non terminée ou comme l'opérateur "défini-ou".
- Vous ne pouvez pas récupérer l'adresse de quelque chose, bien que l'opérateur barre oblique inverse de Perl vous permette de faire quelque chose d'assez proche : créer une référence.
- ARGV doit être en majuscules. \$ARGV[0] est l'argv[1] du C, et argv[0] atterrit dans \$0.
- Les appels systèmes tels que `link()`, `unlink()`, `rename()`, etc. renvoient autre chose que zéro en cas de succès. (En revanche, `system()` retourne un zéro en cas de succès.)
- Les handlers de signaux manipulent des noms de signaux, pas des nombres. Utilisez `kill -1` pour déterminer leurs noms sur votre système.

## 2.3 Pièges de sed

Les programmeurs `sed` expérimentés devraient prendre note de ce qui suit :

- Un programme Perl est exécuté une seule fois et non pour chacune des lignes d'entrée. Vous pouvez obtenir une boucle implicite en utilisant les options `-n` et `-p`.
- Les références arrières dans les substitutions utilisent "\$" à la place de "\".
- Les métacaractères d'expressions rationnelles "(", ")", et "|" ne sont pas précédés de barres obliques inverses.
- L'opérateur d'intervalle est `..`, à la place de la virgule.

## 2.4 Pièges du shell

les programmeurs shell dégourdis devraient prendre note de ce qui suit :

- L'opérateur accent grave effectue une interpolation de variable sans se soucier de la présence ou non d'apostrophes dans la commande.
- L'opérateur accent grave ne fait pas de traduction de la valeur de retour, contrairement au `cs`.
- Les shells (en particulier `cs`) effectuent plusieurs niveaux de substitution sur chaque ligne de commande. Perl ne fait de substitution que dans certaines structures comme les guillemets, les accents graves, les crochets et les motifs de recherche.
- Les shells interprètent les scripts petits bouts par petits bouts. Perl compile tout le programme avant de l'exécuter (sauf les blocs `BEGIN`, qu'il exécute lors de la compilation).
- Les arguments sont disponibles via `@ARGV`, et pas `$1`, `$2`, etc.
- L'environnement n'est pas automatiquement mis à disposition sous forme de variables scalaires distinctes.
- Les tests en shell utilisent "=", "!=", "<", etc. pour les comparaisons de chaînes et "-eq", "-ne", "-lt", etc. pour les comparaisons numériques. En Perl, c'est exactement le contraire : on utilise `eq`, `ne`, `lt`, etc. pour les comparaisons de chaînes et `==`, `!=`, `<`, etc. pour les comparaisons numériques.

## 2.5 Pièges de Perl

Les programmeurs Perl pratiquants devraient prendre note de ce qui suit :

- Souvenez-vous que de nombreuses opérations se comportent différemment selon qu'elles sont appelées dans un contexte de liste ou dans un contexte scalaire. Voir le manuel *perldata* pour plus de détails.
- Évitez autant que possible les barewords, en particulier ceux en minuscules. Vous ne pouvez pas dire rien qu'en regardant si un bareword est une fonction ou une chaîne. En utilisant les apostrophes pour les chaînes et des parenthèses pour les appels de fonctions, vous permettrez qu'on ne les confonde jamais.
- Vous ne pouvez pas distinguer, par simple inspection, les fonctions intégrées qui sont, par construction, des opérateurs unaires (comme `chop()` et `chdir()`) de celles qui sont des opérateurs de liste (comme `print()` et `unlink()`). (Sauf à utiliser des prototypes, les sous-programmes définis par l'utilisateur ne peuvent être **que** des opérateurs de liste, jamais des opérateurs unaires). Voir le manuel *perlop* et le manuel *perlsub*.
- Les gens ont du mal à se souvenir que certaines fonctions utilisent par défaut `$_`, ou `@ARGV`, ou autre chose, tandis que d'autres ne le font pas alors qu'on pourrait s'y attendre.
- La structure `<FH>` n'est pas le nom du handle de fichier, c'est un opérateur lisant une ligne sur ce handle. Les données lues sont affectées à `$_` seulement si la lecture du fichier est la seule condition d'une boucle `while` :

```

while (<FH>      { }
while (defined($_ = <FH>)) { }..
<FH>; # données rejetées !

```

Souvenez-vous de ne pas utiliser = lorsque vous avez besoin de =~ ; ces deux structures sont très différentes :

- \$x = /foo/;
- \$x =~ /foo/;
- La structure do {} n'est pas une véritable boucle sur laquelle vous pourriez utiliser les instructions de contrôle de boucle.
- Utilisez my () pour les variables locales chaque fois que vous pouvez vous en satisfaire (mais voyez le manuel *perldata* pour les cas où vous ne le pouvez pas). L'utilisation de local () donne vraiment une valeur locale à une variable globale, ce qui vous met à la merci d'effets secondaires de portée dynamique imprévus.
- Si vous localisez une variable exportée dans un module, sa valeur exportée ne changera pas. Le nom local devient un alias pour une nouvelle valeur, mais le nom externe est toujours un alias de l'original.

## 2.6 Pièges entre Perl4 et Perl5

Les programmeurs Perl4 pratiquants devraient prendre note des pièges suivants, spécifiques au passage entre Perl4 et Perl5.

Ils sont crûment commandés par la liste suivante :

### Pièges liés aux corrections de bugs, aux désapprobations et aux abandons

Tout ce qui a été corrigé en tant que bug de perl4, supprimé ou désapprouvé en tant que caractéristique de perl4, avec l'intention d'encourager l'usage d'une autre caractéristique de perl5.

### Pièges de l'analyse syntaxique

Pièges qui semblent provenir du nouvel analyseur.

### Pièges numériques

Pièges liés aux opérateurs numériques ou mathématiques.

### Pièges des types généraux de données

Pièges impliquant les types de données standards de perl.

### Pièges du contexte - contexte scalaire et de liste

Pièges liés au contexte dans les instructions et déclarations scalaires ou de liste.

### Pièges de la précedence (les priorités)

Pièges liés à la précedence lors de l'analyse, de l'évaluation et de l'exécution du code.

### Pièges des expressions rationnelles générales lors de l'utilisation de s///, etc.

Pièges liés à l'utilisation de la reconnaissance de motifs.

### Pièges des sous-programmes, des signaux, des tris

Pièges liés à l'utilisation des signaux et des handlers de signaux, aux sous-programmes généraux, et aux tris, ainsi qu'aux sous-programmes de tri.

### Pièges du système d'exploitation

Pièges spécifiques au système d'exploitation.

### Pièges de DBM

Pièges spécifiques à l'utilisation de dbmopen (), et aux implémentations particulières de dbm.

### Pièges non classés

Tout le reste.

Si vous trouvez un exemple de piège de conversion qui ne soit pas listé ici, soumettez-le à <perlbug@perl.org> en vue de son inclusion. Notez aussi qu'un certain nombre d'entre eux peuvent être détectés par le pragma use warnings ou par l'option -w.

## 2.7 Pièges liés aux corrections de bugs, aux désapprobations et aux abandons

Tout ce qui a été abandonné, désapprouvé ou corrigé comme étant un bug depuis perl4.

- Les symboles commençant par "\_" ne sont plus forcément dans main.
- Les symboles commençant par "\_" ne sont plus forcément dans le paquetage main, sauf pour \$\_ lui-même (ainsi que @\_, etc.).

```

package test;
$_legacy = 1;
package main;
print "\$_legacy is ", $_legacy, "\n";
# perl4 affiche : $_legacy is 1
# perl5 affiche : $_legacy is

```

- Le double deux-points dans un nom de variable est un séparateur de nom de paquetage

Le double deux-points est désormais un séparateur de paquetage valide dans un nom de variable. Ainsi ceux-ci se comportent différemment en perl4 et en perl5, car le paquetage n'existe pas.

```

$a=1;$b=2;$c=3;$var=4;
print "$a::$b::$c ";
print "$var::abc::xyz\n";
# perl4 affiche : 1::2::3 4::abc::xyz
# perl5 affiche : 3

```

Étant donné que `::` est maintenant le délimiteur de paquetage préféré, on peut se demander si ceci doit être considéré comme un bug ou non (l'ancien délimiteur de paquetage, `'`, est utilisé ici).

```

$x = 10 ;
print "x=${'x}\n" ;
# perl4 affiche : x=10
# perl5 affiche : Can't find string terminator "'" anywhere before EOF

```

Vous pouvez éviter ce problème, en restant compatible avec perl4, si vous incluez toujours explicitement le nom du paquetage :

```

$x = 10 ;
print "x=${main'x}\n" ;

```

Voyez aussi les pièges de précedence, pour l'analyse de `$.:`

- Les deuxièmes et troisièmes arguments de `splice()` ont un contexte scalaire

Les deuxièmes et troisièmes arguments de `splice()` sont désormais évalués dans un contexte scalaire (comme l'indique le Camel) plutôt que dans un contexte de liste.

```

sub sub1{return(0,2) }          # retourne une liste de 2 éléments
sub sub2{ return(1,2,3)}      # retourne une liste de 3 éléments
@a1 = ("a","b","c","d","e");
@a2 = splice(@a1,&sub1,&sub2);
print join(' ',@a2),"\n";
# perl4 affiche : a b
# perl5 affiche : c d e

```

- Impossible de faire un `goto` vers un bloc optimisé

Vous ne pouvez pas faire un `goto` dans un bloc optimisé. Mince.

```

goto marker1;
for(1){
marker1:
    print "Here I is!\n";
}
# perl4 affiche : Here I is!
# perl5 erreur : Can't "goto" into the middle of a foreach loop

```

- Impossible d'utiliser l'espace dans un nom de variable ou comme délimiteur

Il n'est plus légal syntaxiquement d'utiliser l'espace comme nom de variable, ou comme délimiteur pour tout type de structure entre apostrophes. Mince et remince.

```

$a = ("foo bar");
$b = q baz ;
print "a is $a, b is $b\n";
# perl4 affiche : a is foo bar, b is baz
# erreur de perl5 : Bareword found where operator expected

```

- `while/if BLOCK BLOCK` a disparu

La syntaxe archaïque `while/if BLOC BLOC` n'est plus supportée.

```

if { 1 } {
    print "True!";
}
else {
    print "False!";
}
# perl4 affiche : True!

```

- ```

# erreur de perl5 : syntax error at test.pl line 1, near "if {"

```
- **\*\* est prioritaire sur le moins unaire**

L'opérateur **\*\*** est désormais plus prioritaire que le moins unaire. Cela devait fonctionner ainsi selon la documentation, mais ce n'était pas le cas.

```

print -4**2, "\n";
# perl4 affiche : 16
# perl5 affiche : -16

```
- **foreach a changé lorsqu'il est appliqué à une liste**

La signification de `foreach{}` a légèrement changé lorsqu'il traverse une liste qui n'est pas un tableau. Auparavant, il fonctionnait en affectant la liste à un tableau temporaire, mais ce n'est plus le cas (pour des raisons d'efficacité). Cela veut dire que vous itérez maintenant sur les vraies valeurs de la liste, pas sur des copies de ces valeurs. Les modifications de la variable de boucle peuvent changer les valeurs originelles.

```

@list = ('ab', 'abc', 'bcd', 'def');
foreach $var (grep(/ab/, @list)){
    $var = 1;
}
print (join(':', @list));
# perl4 affiche : ab:abc:bcd:def
# perl5 affiche : 1:1:bcd:def

```

Pour garder la sémantique de Perl4, vous devrez affecter explicitement votre liste à un tableau temporaire puis itérer dessus. Vous pourriez par exemple devoir changer

```

foreach $var (grep(/ab/, @list)){
en
    foreach $var (@tmp = grep(/ab/, @list)){

```

Sinon changer `$var` modifiera les valeurs de `@list` (Cela arrive le plus souvent lorsque vous utilisez `$_` comme variable de boucle, et appelez dans celle-ci des sous-programmes qui ne localisent pas correctement `$_`).
- **split sans argument a changé de comportement**

`split` sans argument se comporte désormais comme `split ' '` (qui ne retourne pas de champ nul initial si `$_` commence par une espace), il se comportait habituellement comme `split /\s+/` (qui le fait).

```

$_ = ' hi mom';
print join(':', split);
# perl4 affiche : :hi:mom
# perl5 affiche : hi:mom

```
- **comportement corrigé de l'option -e**

Perl 4 ignorait tout texte attaché à une option **-e**, et prenait toujours le petit bout de code dans l'argument suivant. Qui plus est, il acceptait silencieusement une option **-e** non suivie d'un argument. Ces deux comportements ont été supprimés.

```

perl -e'print "attached to -e" 'print "separate arg"
# perl4 affiche : separate arg
# perl5 affiche : attached to -e
perl -e
# perl4 affiche :
# perl5 meurt : No code specified for -e.

```
- **push retourne le nombre d'éléments de la liste résultante**

En Perl 4, la valeur de retour de `push` n'était pas documentée, mais c'était en fait la dernière valeur poussée sur la liste cible. En Perl 5, la valeur de retour de `push` est documentée, mais elle a changé, c'est désormais le nombre d'éléments de la liste résultante.

```

@x = ('existing');
print push(@x, 'first new', 'second new');
# perl4 affiche : second new
# perl5 affiche : 3

```
- **Certains messages d'erreur ont changé**

Certains messages d'erreur sont différents.
- **split respecte les arguments des sous-programmes**

En Perl 4, dans un contexte de liste, si les délimiteurs du premier argument de `split()` étaient ??, le résultat était placé dans `@_` tout en étant retourné normalement. Perl 5 respecte maintenant les arguments de vos sous-programmes.
- **Bogues supprimés**

Certains bugs ont peut-être été supprimés par inadvertance. :-)

## 2.8 Pièges de l'analyse syntaxique

Pièges entre Perl4 et Perl5 ayant un rapport avec l'analyse syntaxique.

- Un espace entre . et = produit une erreur

Notez l'espace entre . et =

```
$string . = "more string";
print $string;
# perl4 affiche: more string
# perl5 affiche : syntax error at - line 1, near ". ="
```

- Meilleure analyse syntaxique en perl 5

L'analyse syntaxique est meilleure en perl 5

```
sub foo {}
&foo
print("hello, world\n");
# perl4 affiche : hello, world
# perl5 affiche : syntax error
```

- analyse syntaxique des fonctions

Règle "si ça a l'air d'une fonction, c'est une fonction".

```
print
($foo == 1) ? "is one\n" : "is zero\n";
# perl4 affiche : is zero
# perl5 avertit : "Useless use of a constant in void context" if using -w
```

- L'interpolation de la construction \$#tableau a changé

L'interpolation de chaîne de la structure \$#tableau diffère lorsque des accolades sont utilisées autour du nom.

```
@a = (1..3);
print "${#a}";
# perl4 affiche : 2
# perl5 échoue sur une erreur de syntaxe
@a = (1..3);
print "$#{a}";
# perl4 affiche : {a}
# perl5 affiche : 2
```

- Perl devine si un { qui suit un map ou un grep débute un BLOC ou une référence à une table de hachage

Lorsque Perl rencontre map { (ou grep {}), il doit deviner si l'accolade { débute un nouveau BLOC ou une référence à une table de hachage. Si il devine mal, il indiquera une erreur de syntaxe au niveau de l'accolade fermante } et de la virgule manquante (ou inattendue).

Utilisez un + unaire avant l'accolade { d'une référence à une table de hachage ou un + unaire appliqué à la première chose qui commence le BLOC (après l'accolade {}) pour que perl devine correctement tout le temps. (Voir le titre map dans le manuel *perlfunc*.)

## 2.9 Pièges numériques

Pièges entre Perl4 et Perl5 ayant un rapport avec les opérateurs numériques, les opérandes, ou leurs sorties.

- Sortie formatée et chiffres significatifs

Sortie formatée et chiffres significatifs. En général, Perl 5 essaye d'être plus précis. Par exemple, sur une Sparc sous Solaris :

```
print 7.373504 - 0, "\n";
printf "%20.18f\n", 7.373504 - 0;
# Perl4 affiche :
7.3735039999999996141
7.373503999999999614
# Perl5 affiche :
7.373504
7.373503999999999614
```

Remarquez que le premier résultat est mieux en Perl 5.

Les résultats que vous obtenez sont variables puisque vos fonctions d'affichage des flottants et vos flottants eux-même peuvent être différents.

- Détection du dépassement de l'entier signé maximale lors d'une auto-incrémentation

Cette détection a été rétablie. Auparavant on pouvait montre que l'opérateur d'auto-incrémentation ne voyait pas qu'un nombre avait dépassé la limite supérieure des entiers signés. Cela a été réglé dans la version 5.003\_04. Mais soyez toujours prudent lorsque vous utilisez de grands entiers. En cas de doute :

```
use Math::BigInt;
```

- L'affectation du résultat d'un test d'égalité numérique ne fonctionne pas

L'affectation du résultat d'un test d'égalité numérique ne fonctionne pas en perl5 lorsque le test échoue (auparavant il renvoyait 0). Les tests logique retourne maintenant une chaîne vide, au lieu de 0.

```
$p = ($test == 1);
print $p, "\n";
# perl4 affiche : 0
# perl5 affiche :
```

Voir aussi §2.13 pour un autre exemple de cette nouvelle caractéristique...

- Opérateurs de chaîne bit par bit

Lorsque des opérateurs bit par bit qui peuvent travailler sur des nombres ou sur des chaînes (& | ^ ~) n'ont que des chaînes pour arguments, perl4 traite les opérandes comme des chaînes de bits dans la mesure où le programme contenait un appel à la fonction `vec()`. perl5 traite les opérandes chaînes comme des chaînes de bits (Voir le titre Opérateurs bit à bit sur les chaînes dans le manuel *perlop* pour plus de détails).

```
$fred = "10";
$barney = "12";
$betty = $fred & $barney;
print "$betty\n";
# Décommentez la ligne suivante pour changer le comportement de perl4
# ($dummy) = vec("dummy", 0, 0);
# Perl4 affiche :
8
# Perl5 affiche :
10
# Si vec() est utilisé quelque part dans le programme, les deux
# affichent :
10
```

## 2.10 Pièges des types de données généraux

Pièges entre Perl4 et Perl5 impliquant la plupart des types de données, et leur usage dans certaines expressions et/ou contextes.

- Les indices négatifs partent maintenant de la fin du tableau

Les indices de tableau négatifs sont maintenant comptés depuis la fin du tableau.

```
@a = (1, 2, 3, 4, 5);
print "The third element of the array is $a[3] also expressed as $a[-2] \n";
# perl4 affiche : The third element of the array is 4 also expressed as
# perl5 affiche : The third element of the array is 4 also expressed as 4
```

- Diminuer la valeur de \$#tableau supprime les éléments du tableau

Désormais, diminuer la valeur de \$#array supprime réellement les éléments excédentaires du tableau, et rend impossible leur récupération.

```
@a = (a,b,c,d,e);
print "Before: ", join(' ', @a);
$a = 1;
print ", After: ", join(' ', @a);
$a = 3;
print ", Recovered: ", join(' ', @a), "\n";
# perl4 affiche : Before: abcde, After: ab, Recovered: abcd
# perl5 affiche : Before: abcde, After: ab, Recovered: ab
```

- Les tables de hachage sont définies même vide

Les hachages sont définis avant même d'être remplie

```
local($s, @a, %h);
die "scalar \$s defined" if defined($s);
die "array \@a defined" if defined(@a);
die "hash \%h defined" if defined(%h);
# perl4 affiche :
# perl5 meurt : hash %h defined
```

Perl génère désormais un avertissement lorsqu'il voit `defined(@a)` et `defined(%h)`.

- Affectation globale de variable localisée à variable

L'affectation globale de variable à variable échouera si la variable affectée est localisée après l'affectation

```
@a = ("This is Perl 4");
*b = *a;
local(@a);
print @b, "\n";
# perl4 affiche : This is Perl 4
# perl5 affiche :
```

#### – Affectation globale de undef

L'affectation globale de `undef` n'a pas d'effet en Perl 5. En Perl 4, elle rend indéfini le scalaire associé (mais peut avoir d'autres effets secondaires, y compris des SEGV). Perl 5 émet un avertissement lorsqu'on affecte globalement `undef` (Notez bien que l'affectation globale de `undef` n'est pas la même chose que d'appliquer la fonction `undef` à une variable globale (`undef $foo`)).

```
$foo = "bar";
*foo = undef;
print $foo;
# perl4 affiche :
# perl4 avertit : "Use of uninitialized variable" si -w
# perl5 affiche : bar
# perl5 avertit : "Undefined value assigned to typeglob" si -w
```

#### – Changements dans la négation unaire (de chaînes)

Changements dans la négation unaire (de chaînes). Ce changement affecte à la fois la valeur de retour et ce qu'elle fait à l'incrément automatique.

```
$x = "aaa";
print ++$x, " : ";
print -$x, " : ";
print ++$x, "\n";
# perl4 affiche : aab : -0 : 1
# perl5 affiche : aab : -aab : aac
```

#### – Modifier une constante est interdit

perl 4 vous laisse modifier les constantes :

```
$foo = "x";
&mod($foo);
for ($x = 0; $x < 3; $x++) {
    &mod("a");
}
sub mod {
    print "before: $_[0]";
    $_[0] = "m";
    print " after: $_[0]\n";
}
# perl4:
# before: x after: m
# before: a after: m
# before: m after: m
# before: m after: m
# Perl5:
# before: x after: m
# Modification of a read-only value attempted at foo.pl line 12.
# before: a
```

#### – Le comportement de `defined $var` a changé

Le comportement est légèrement différent pour :

```
print "$x", defined $x
# perl 4: 1
# perl 5: <no output, $x is not called into existence>
```

#### – Suicide de variable

Le suicide de variable est plus cohérent sous Perl 5. Perl5 montre le même comportement pour les hachages et les scalaires, que celui montré par Perl4 uniquement pour les scalaires.

```
$aGlobal{ "aKey" } = "global value";
print "MAIN:", $aGlobal{"aKey"}, "\n";
$GlobalLevel = 0;
&test( *aGlobal );
sub test {
```

```

local( *theArgument ) = @_;
local( %aNewLocal ); # perl 4 != 5.0011,m
$aNewLocal{"aKey"} = "this should never appear";
print "SUB: ", $theArgument{"aKey"}, "\n";
$aNewLocal{"aKey"} = "level $GlobalLevel"; # what should print
$GlobalLevel++;
if( $GlobalLevel<4 ) {
    &test( *aNewLocal );
}
}
# Perl4:
# MAIN:global value
# SUB: global value
# SUB: level 0
# SUB: level 1
# SUB: level 2
# Perl5:
# MAIN:global value
# SUB: global value
# SUB: this should never appear
# SUB: this should never appear
# SUB: this should never appear

```

## 2.11 Pièges du contexte - contextes scalaires et de liste

\*\*\*\*\*

- Les éléments des liste d'arguments des formats sont évalués dans un contexte de liste  
Les éléments des listes d'arguments pour les formats sont désormais évalués dans un contexte de liste. Cela signifie que vous pouvez maintenant interpoler les valeurs de liste.

```

@fmt = ("foo", "bar", "baz");
format STDOUT=
@<<<<< @| | | | @>>>>>
@fmt;

```

```

.
write;
# erreur de perl4 : Please use commas to separate fields in file
# perl5 affiche : foo bar baz

```

- caller retourne faux si il est évalué dans un context scalaire et qu'il n'y a pas d'appelant  
La fonction caller() retourne maintenant la valeur faux dans un contexte scalaire s'il n'y a pas d'appelant. Cela laisse les fichiers de bibliothèque déterminer s'ils sont en train d'être demandés.

```

caller() ? (print "You rang?\n") : (print "Got a 0\n");
# erreur de perl4 : There is no caller
# perl5 affiche : Got a 0

```

- L'opérateur virgule dans un contexte scalaire donne un contexte scalaire à ses arguments  
L'opérateur virgule dans un contexte scalaire est désormais garanti comme donnant un contexte scalaire à ses arguments.

```

@y= ('a', 'b', 'c');
$x = (1, 2, @y);
print "x = $x\n";
# Perl4 affiche : x = c # pense contexte de liste,
# interpole la liste
# Perl5 affiche : x = 3 # sait que le scalaire utilise la
# longueur de la liste

```

- sprintf() est prototypé comme (\$;@)  
La fonction sprintf() utilise maintenant le prototype (\$;@). Elle donne donc un contexte scalaire à son premier argument. Donc si vous lui passer un tableau, elle ne fera probablement pas ce que vous souhaitez contrairement à Perl 4:

```

@z = ('%s%s', 'foo', 'bar');
$x = sprintf(@z);
print $x;
# perl4 affiche : foobar

```

```

# perl5 affiche : 3
printf() continue à fonctionner comme en Perl 4. Donc :
@z = ('%s%s', 'foo', 'bar');
printf STDOUT (@z);
# perl4 affiche : foobar
# perl5 affiche : foobar

```

## 2.12 Pièges de la précedence (les priorités)

Pièges entre Perl4 et Perl5 impliquant l'ordre de précedence.

Perl 4 a presque les mêmes règles de précedence que Perl 5 pour les opérateurs qu'ils ont en commun. Toutefois, Perl 4 semble avoir contenu des incohérences ayant rendu son comportement différent de celui qui était documenté.

### – LHS vs. RHS pour tout opérateur d'affectation

LHS vs. RHS pour tout opérateur d'affectation. LHS est d'abord évalué en perl4, mais en second en perl5 ; ceci peut modifier les relations entre les effets de bord dans les sous-expressions.

```

@arr = ( 'left', 'right' );
$a{shift @arr} = shift @arr;
print join( ' ', keys %a );
# perl4 affiche : left
# perl5 affiche : right

```

### – Erreurs sémantiques dues aux règles de précedence

Voici des expressions devenues des erreurs sémantiques à cause de la précedence :

```

@list = (1,2,3,4,5);
%map = ("a",1,"b",2,"c",3,"d",4);
$n = shift @list + 2; # premier élément de la liste plus 2
print "n is $n, ";
$m = keys %map + 2; # nombre d'éléments du hachage plus 2
print "m is $m\n";
# perl4 affiche : n is 3, m is 6
# erreur de perl5 et échec de la compilation

```

### – Précedence identique pour les opérateurs d'affectation et l'affectation

La précedence des opérateurs d'affectation est désormais la même que celle de l'affectation. Perl 4 leur donnait par erreur la précedence de l'opérateur associé. Vous devez donc maintenant les mettre entre parenthèses dans les expressions telles que

```

/foo/ ? ($a += 2) : ($a -= 2);

```

Autrement

```

/foo/ ? $a += 2 : $a -= 2

```

serait analysé de façon erronée sous la forme

```

(/foo/ ? $a += 2 : $a) -= 2;

```

D'autre part,

```

$a += /foo/ ? 1 : 2;

```

fonctionne désormais comme un programmeur C pourrait s'y attendre.

### – open peut nécessiter des parenthèse autour du filehandle

`open FOO || die;` est désormais incorrect. Vous devez mettre le handle de fichier entre parenthèses. Sinon, perl5 laisse sa précedence par défaut à l'instruction :

```

open(FOO || die);
# perl4 ouvre ou meurt
# perl5 ouvre FOO, et ne mourrait que si 'FOO' était faux, c.-à-d. jamais

```

### – Suppression de la priorité de \$: sur C\$: :

perl4 donne la précedence à la variable spéciale \$:, tandis que perl5 traite \$::: comme étant le paquetage principale

```

$a = "x"; print "$::a";
# perl 4 affiche : -:a
# perl 5 affiche : x

```

### – Précedence des opérateurs de test sur fichiers telle que documentée

perl4 a une précedence bugguée pour les opérateurs de test de fichiers vis-à-vis des opérateurs d'affectation. Ainsi, bien que la table de précedence de perl4 laisse à penser que `-e $foo .= "q"` devrait être analysé comme `((-e $foo) .= "q")`, il l'analyse en fait comme `(-e ($foo .= "q"))`. En perl5, la précedence est telle que documentée.

```

-e $foo .= "q"

```

- ```

# perl4 affiche : no output
# perl5 affiche : Can't modify -e in concatenation

```
- `keys`, `each`, `values` sont des opérateurs unaires nommés normaux  
En perl4, `keys()`, `each()` et `values()` étaient des opérateurs spéciaux de haute priorité qui agissaient sur un simple hachage, mais en perl5, ce sont des opérateurs unaires nommés normaux. Tels que documentés, ces opérateurs ont une priorité plus basse que les opérateurs arithmétiques et de concaténation `+` `-` `..`, mais les variantes de perl4 de ces opérateurs se lient en fait plus étroitement que `+` `-` `..`. Ainsi, pour :

```

%foo = 1..10;
print keys %foo - 1
# perl4 affiche : 4
# perl5 affiche : Type of arg 1 to keys must be hash (not subtraction)

```

Le comportement de perl4 était probablement plus utile, mais moins cohérent.

## 2.13 Pièges des expressions rationnelles générales lors de l'utilisation de `s///`, etc.

Tous types de pièges concernant les expressions rationnelles.

- `s' $lhs' $rhs'` n'effectue d'interpolation ni d'un côté ni de l'autre  
`s' $lhs' $rhs'` n'effectue désormais d'interpolation ni d'un côté ni de l'autre. `$lhs` était habituellement interpolé, mais pas `$rhs` (et ne détecte toujours pas un `'$'` littéral dans la chaîne)

```

$a=1;$b=2;
$string = '1 2 $a $b';
$string =~ s'$a'$b';
print $string, "\n";
# perl4 affiche : $b 2 $a $b
# perl5 affiche : 1 2 $a $b

```
- `/m/g` attache son état à la chaîne fouillée  
`m/g` attache maintenant son état à la chaîne fouillée plutôt que l'expression rationnelle (une fois que la portée d'un bloc est quittée pour le sous-programme, l'état de la chaîne fouillée est perdu)

```

$_ = "ababab";
while(m/ab/g){
    &doit("blah");
}
sub doit{local($_) = shift; print "Got $_ "}
# perl4 affiche : Got blah Got blah Got blah Got blah
# perl5 affiche : boucle infinie de blah...

```
- `m//o` utilisé dans un sous-programme anonyme  
Couramment, si vous utilisez le qualificateur `m//o` sur une expression rationnelle dans un sous-programme anonyme, toutes les fermetures générées dans ce sous-programme anonyme utiliseront l'expression rationnelle telle qu'elle a été compilée lors de sa toute première utilisation dans une telle fermeture. Par exemple, si vous dites

```

sub build_match {
    my($left,$right) = @_;
    return sub { $_[0] =~ /$left stuff $right/o; };
}
$good = build_match('foo','bar');
$bad = build_match('baz','blarch');
print $good->('foo stuff bar') ? "ok\n" : "not ok\n";
print $bad->('baz stuff blarch') ? "ok\n" : "not ok\n";
print $bad->('foo stuff bar') ? "not ok\n" : "ok\n";

```

Pour la plupart des distributions de Perl5, cela affichera :

```

ok
not ok
not ok

```

`build_match()` retournera toujours un sous-programme qui correspondra au contenu de `$left` et de `$right` tels qu'ils étaient la première fois que `build_match()` a été appelé, et pas tels qu'ils sont dans l'appel en cours.
- `$+` ne contient plus la chaîne reconnue  
Si aucune parenthèse n'est utilisée dans une correspondance, Perl4 fixe `$+` à toute la correspondance, tout comme `$&`. Perl5 ne le fait pas.

```

"abcdef" =~ /b.*e/;
print "\$+ = $+\n";
# perl4 affiche : bcde

```

```
# perl5 affiche :
```

- Une substitution retourne la chaîne vide si elle échoue

La substitution retourne désormais la chaîne vide si elle échoue

```
$string = "test";
$value = ($string =~ s/foo//);
print $value, "\n";
# perl4 affiche : 0
# perl5 affiche :
```

Voir aussi le titre Pièges Numériques dans ce document pour un autre exemple de cette nouvelle caractéristique.

- s`lhs`rhs` est une substitution normale

s`lhs`rhs` (en utilisant des accents graves) est maintenant une substitution normale, sans expansion des accents graves

```
$string = "";
$string =~ s``hostname`;
print $string, "\n";
# perl4 affiche : <the local hostname>
# perl5 affiche : hostname
```

- Analyse stricte des variables utilisées dans les expressions rationnelles

Analyse plus stricte des variables utilisées dans les expressions rationnelles

```
s/^([$grpc]*$grpc[$opt$plus$rep]?)/o;
# perl4: compile sans erreur
# perl5: compile avec l'erreur :
# Scalar found where operator expected ..., near "$opt$plus"
```

un ajout à cet exemple, apparemment depuis le même script, est la véritable valeur de la chaîne après la substitution.

[*\$opt*] est une classe de caractère en perl4 et un indice de tableau en perl5

```
$grpc = 'a';
$opt = 'r';
$_ = 'bar';
s/^([$grpc]*$grpc[$opt]?)/foo/;
print ;
# perl4 affiche : foo
# perl5 affiche : foobar
```

- m?x? ne correspond qu'une seule fois

Sous perl5, m?x? ne correspond qu'une fois, comme ?x?. Sous perl4, cela correspondait plusieurs fois, comme /x/ ou m!x!.

```
$test = "once";
sub match { $test =~ m?once?; }
&match();
if( &match() ) {
    # m?x? correspond plusieurs fois
    print "perl4\n";
} else {
    # m?x? correspond une seule fois
    print "perl5\n";
}
# perl4 affiche : perl4
# perl5 affiche : perl5
```

- L'échec d'une reconnaissance par expression rationnelle ne réinitialise pas les variables de reconnaissance

Contrairement à Ruby, l'échec d'une reconnaissance par expression rationnelle ne réinitialise pas les variables de reconnaissance (\$1, \$2, ..., \$', ...).

## 2.14 Pièges des sous-programmes, des signaux et des tris

Le groupe général de pièges entre Perl4 et Perl5 ayant un rapport avec les Signaux, les Tris, et leurs sous-programmes associés, ainsi que les pièges généraux liés aux sous-programmes. Cela inclut certains pièges spécifiques au système d'exploitation.

- Les barewords (mots simples) sont considérés comme des appels de sous-programmes

Les barewords (mots simples) qui étaient considérés comme des chaînes pour Perl sont maintenant considérés comme des appels de sous-programmes si un sous-programme ayant ce nom est déjà défini (le compilateur l'a déjà vu).

```
sub SeeYa { warn"Hasta la vista, baby!" }
```

```

$SIG{'TERM'} = SeeYa;
print "SIGTERM is now $SIG{'TERM'}\n";
# perl4 affiche : SIGTERM is now main'SeeYa
# perl5 affiche : SIGTERM is now main::1

```

Utilisez **-w** pour capturer celui-ci

- `reverse` n'est plus autorisé en tant que nom de sous-programme de tri  
`reverse` n'est plus autorisé en tant que nom de sous-programme de tri.

```

sub reverse{ print "yup "; $a <=> $b }
print sort reverse (2,1,3);
# perl4 affiche : yup yup 123
# perl5 affiche : 123
# perl5 avertit (si -w) : Ambiguous call resolved as CORE::reverse()

```

- `warn()` ne vous laisse pas spécifier un handle de fichier.

Bien qu'il ait `_toujours_` imprimé sur `STDERR`, `warn()` vous laissait spécifier un handle de fichier en perl4. Avec perl5, ce n'est plus le cas.

```

warn STDERR "Foo!";
# perl4 affiche : Foo!
# perl5 affiche : String found where operator expected

```

## 2.15 Pièges du système d'exploitation

- SysV réinitialise correctement un gestionnaire de signal (signal handler)

Sous HPUX, et certains autres systèmes d'exploitation de la famille SysV, on devait réinitialiser tout handle de signal à l'intérieur de la fonction de gestion du signal, chaque fois qu'un signal était traité avec perl4. Avec perl5, la réinitialisation est désormais faite correctement. Tout code reposant sur un handler n'étant `_pas_` réinitialisé devra être réécrit.

Depuis la version 5.002, Perl utilise `sigaction()` sous SysV.

```

sub gotit {
    print "Got @_... ";
}
$SIG{'INT'} = 'gotit';
$| = 1;
$pid = fork;
if ($pid) {
    kill('INT', $pid);
    sleep(1);
    kill('INT', $pid);
} else {
    while (1) {sleep(10);}
}
# perl4 (HPUX) affiche : Got INT...
# perl5 (HPUX) affiche : Got INT... Got INT...

```

- La fonction `seek()` sous SysV ajoute maintenant correctement

Sous les systèmes d'exploitation de la famille SysV, `seek()` sur un fichier ouvert en mode d'ajout » fait désormais ce qu'il doit selon la page de manuel de `fopen()`. C'est-à-dire que lorsqu'un fichier est ouvert en mode d'ajout, il est impossible d'écraser les informations déjà présentes dans le fichier.

```

open(TEST, ">>seek.test");
$start = tell TEST ;
foreach(1 .. 9){
    print TEST "$_ ";
}
$end = tell TEST ;
seek(TEST, $start, 0);
print TEST "18 characters here";
# perl4 (solaris) seek.test dit : 18 characters here
# perl5 (solaris) seek.test dit : 1 2 3 4 5 6 7 8 9 18 characters here

```

## 2.16 Pièges de l'interpolation

Pièges entre Perl4 et Perl5 ayant un rapport avec la façon dont les choses sont interpolées dans certaines expressions, instructions, dans certains contextes, ou quoi que ce soit d'autre.

- Un @ est désormais toujours interpolé en tant que tableau dans une chaîne entre guillemets  
Un @ est désormais toujours interpolé en tant que tableau dans une chaîne entre guillemets
 

```
print "To: someone@somewhere.com\n";
# perl4 affiche : To:someone@somewhere.com
# perl < 5.6.1, erreur : In string, @somewhere now must be written as \@somewhere
# perl >= 5.6.1, avertissement : Possible unintended interpolation of @somewhere in string
```

- Une chaîne entre guillemets ne peut plus se terminer par un \$ sans échappement  
Les chaînes entre guillemets ne peuvent plus se terminer par un \$ non protégé par le caractère d'échappement.

```
$foo = "foo$";
print "foo is $foo\n";
# perl4 affiche : foo is foo$
# erreur de perl5 : Final $ should be \$ or $name
```

Note: perl5 NE génère PAS une erreur pour un @ final.

- Une expression quelconque entre accolades dans une chaîne entre guillemets est évaluée  
Perl évalue maintenant parfois des expressions quelconques placées entre accolades qui apparaissent entre des guillemets (habituellement lorsque l'accolade ouvrante est précédée par \$ ou @).

```
@www = "buz";
$foo = "foo";
$bar = "bar";
sub foo { return "bar" };
print "|@{w.w.w}|${main'foo}|";
# perl4 affiche : @{w.w.w}|foo|
# perl5 affiche : |buz|bar|
```

Notez que vous pouvez utiliser `use strict;` pour vous mettre à l'abri de tels pièges sous perl5.

- \$\$x tente de déréférencer \$x  
La construction "this is \$\$x" interpolait le pid à cet endroit, mais elle essaye maintenant de déréférencer \$x. \$\$ tout seul fonctionne toutefois toujours bien.

```
$s = "a reference";
$x = *s;
print "this is $$x\n";
# perl4 affiche : this is XXXx (XXX is the current pid)
# perl5 affiche : this is a reference
```

- Le création d'une table de hachage au vol par un eval "EXPR" doit être protégée  
La création de hachage à la volée par eval "EXPR" exige maintenant soit que les deux \$ soient protégés dans la spécification du nom du hachage, soit que les deux accolades soient protégées. Si les accolades sont protégées, le résultat sera compatible entre perl4 et perl5. C'est une pratique très commune qui devrait être modifiée de façon à utiliser si possible la forme bloc d'eval{ }.

```
$hashname = "foobar";
$key = "baz";
$value = 1234;
eval "\$$hashname{'$key'} = q|$value|";
(defined($foobar{'baz'})) ? (print "Yup") : (print "Nope");
# perl4 affiche : Yup
# perl5 affiche : Nope
```

En changeant

```
eval "\$$hashname{'$key'} = q|$value|";
```

par

```
eval "\$\$hashname{'$key'} = q|$value|";
```

on obtient le résultat suivant :

```
# perl4 affiche : Nope
# perl5 affiche : Yup
```

ou en le changeant par

```
eval "\$$hashname\{'$key'\} = q|$value|";
```

on obtient le résultat suivant :

```
# perl4 affiche : Yup
# perl5 affiche : Yup
# et est compatible avec les deux versions
```

- Bogues des versions précédentes de perl

Certains programmes perl4 se reposaient inconsciemment sur les bugs de versions précédentes de perl.

```
perl -e '$bar=q/not/; print "This is $foo{$bar} perl5"'
# perl4 affiche : This is not perl5
```

```
# perl5 affiche : This is perl5
```

- Interpolation de crochets (tableaux) et des accolades (tables de hachage)

Vous devez faire attention à l'interpolation des crochets (tableaux) et des accolades (tables de hachage).

```
print "$foo["
perl 4 affiche : [
perl 5 affiche : syntax error
print "$foo{"
perl 4 affiche : {
perl 5 affiche : syntax error
```

Perl 5 s'attend à trouver un indice (ou une clé) après le crochet ouvrant (ou l'accolade ouvrante) ainsi que le crochet fermant (l'accolade fermante) correspondant. Pour retrouver le comportement de Perl 4, vous devez protéger votre crochet (ou accolade) par le caractère d'échappement.

```
print "$foo\[";
print "$foo\{";
```

- Interpolation de `\$$foo{bar}`

De façon similaire, prenez garde à `\$$foo{bar}`

```
$foo = "baz";
print "\$$foo{bar}\n";
# perl4 affiche : $baz{bar}
# perl5 affiche : $
```

Perl 5 cherche `$array{bar}` qui n'existe pas, mais perl 4 est content de simplement substituer `$foo` par "baz". Faites attention à cela, en particulier dans les `eval`'s.

- Une chaîne `qq()` dans un `eval` ne trouve pas sa fin de chaîne

Une chaîne `qq()` passée à `eval` pose le problème suivant :

```
eval qq(
    foreach \y (keys %\x\ ) {
        \ycount++;
    }
);
# perl4 exécute cela sans problème
# perl5 affiche : Can't find string terminator ")"
```

## 2.17 Pièges DBM

Pièges généraux de DBM.

- Perl5 doit être lié avec la même bibliothèque `dbm/ndbm` que la version par défaut de `dbmopen()`

Les bases de données existantes créées sous perl4 (ou tout autre outil `dbm/ndbm`) peuvent faire échouer le même script une fois exécuté sous perl5. L'exécutable de perl5 doit avoir été lié avec le même `dbm/ndbm` par défaut que `dbmopen()` pour qu'il fonctionne correctement sans lien via `tie` vers une implémentation `dbm` sous forme d'extension.

```
dbmopen(%dbm, "file", undef);
print "ok\n";
# perl4 affiche : ok
# perl5 affiche : ok (IFF linked with -ldb or -lndbm)
```

- Un dépassement de la limite de taille d'une clé ou d'une valeur DBM arrête perl5 immédiatement

Les bases de données existantes créées sous perl4 (ou tout autre outil `dbm/ndbm`) peuvent faire échouer le même script une fois exécuté sous perl5. L'erreur générée lorsque la limite de la taille clé/valeur est dépassée provoquera la sortie immédiate de perl5.

```
dbmopen(DB, "testdb", 0600) || die "couldn't open db! $!";
$DB{'trap'} = "x" x 1024; # valeur trop grande pour la plupart
                        # des dbm/ndbm

print "YUP\n";
# perl4 affiche :
dbm store returned -1, errno 28, key "trap" at - line 3.
YUP
# perl5 affiche :
dbm store returned -1, errno 28, key "trap" at - line 3.
```

## 2.18 Pièges non classés

Tout le reste.

- Piège `require/do` utilisant la valeur de retour

Si le fichier `doit.pl` contient :

```
sub foo {
    $rc = do "./do.pl";
    return 8;
}
print &foo, "\n";
```

Et le fichier `do.pl` contient l'unique ligne suivante :

```
return 3;
```

Exécuter `doit.pl` donne le résultat suivant :

```
# perl 4 affiche : 3 (abandonne tôt le sous-programme)
# perl 5 affiche : 8
```

Le comportement est le même si vous remplacez `do` par `require`.

- `split` sur une chaîne vide avec une `LIMIT` spécifiée

```
$string = '';
@list = split(/foo/, $string, 2)
```

Perl4 retourne une liste à un élément contenant la chaîne vide mais Perl5 retourne une liste vide.

Comme toujours, si certains de ces pièges sont déclarés un jour officiellement comme étant des bugs, ils seront corrigés et retirés.

## 3 TRADUCTION

### 3.1 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/>.

### 3.2 Traducteur

Traduction initiale (5.6.0) : Roland Trique <[roland.trique@uhb.fr](mailto:roland.trique@uhb.fr)>. Mise à jour (5.8.8) : Paul Gaborit <[paul.gaborit@enstimac.fr](mailto:paul.gaborit@enstimac.fr)>.

### 3.3 Relecture

Personne pour l'instant.

## 4 À propos de ce document

Ce document est la traduction française du document original 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 par 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.

*Ce document PDF est distribué selon les termes de la license Artistique de Perl. Toute autre distribution de ce fichier ou de ses dérivés impose qu'un arrangement soit fait avec le(s) propriétaire(s) des droits. Ces droits appartiennent aux auteurs du document original (lorsqu'ils sont identifiés dans la rubrique AUTEUR), aux traducteurs et relecteurs pour la version française et à moi-même pour la version PDF.*