perlvar

Table des matières

1 NAME/NOM

perlvar - Variables prédéfinies en Perl

2 DESCRIPTION

2.1 Noms prédéfinis

Les noms suivants ont une signification spéciale en Perl. La plupart de ces noms ont des mnémoniques acceptables ou équivalents dans l'un des shells. Néanmoins, si vous souhaitez utiliser des descripteurs longs, vous avez juste à ajouter

```
use English;
```

en tête de votre programme. Cela créera un alias entre les noms courts et les noms longs du module courant. Certains ont même des noms de longueur intermédiaire, généralement empruntés à **awk**. En général, il est préférable d'invoquer

```
use English '-no_match_vars';
```

si vous n'avez pas besoin de $PREMATCH, $MATCH, ou $POSTMATCH, ce qui évite une baisse de performance certaine dans le traitement des expressions rationnelles. Voir *English*.

Les variables dépendant du descripteur de fichier courant peuvent être initialisées en appelant une méthode de l'objet `IO::Handle`, bien que ce soit moins efficace que d'utiliser les variables intégrées courantes. (Pour cela, les sous-titres ci-dessous contiennent le mot HANDLE). Vous devez écrire d'abord :

```
use IO::Handle;
```

après quoi vous pouvez utiliser soit

```
method HANDLE EXPR
```

soit, de manière plus sûre,

```
HANDLE->method(EXPR)
```

Chaque méthode retourne l'ancienne valeur de l'attribut `IO::Handle`. Les méthodes acceptent chacune EXPR en option, qui, s'il est précisé, spécifie la nouvelle valeur pour l'attribut `IO::Handle` en question. S'il n'est pas précisé, la plupart des méthodes ne modifient pas la valeur courante, exceptée `autoflush()`, qui fixera la valeur à 1, juste pour se distinguer.

Parce que le chargement de la classe `IO::Handle` est coûteux, vous devriez apprendre à utiliser les variables intégrées normales.

Quelques-unes de ces variables sont considérées comme étant en «lecture seule». Cela signifie que si vous essayez de leur attribuer une valeur, directement ou indirectement à travers une référence, vous obtiendrez une erreur d'exécution.

Vous devriez être très prudent quand vous modifiez les valeurs par défaut de la plupart des variables spéciales décrites dans ce document. Dans la plupart des cas vous devez localiser ces variables avant de les changer, puisque, si vous ne faites pas, le changement peut affecter d'autres modules qui comptent sur les valeurs par défaut des variables spéciales que vous avez changées. Voici une des façons correctes de lire un fichier entier en une seule fois :

```
open my $fh, "foo" or die $!;
local $/; # établit le mode slurp en local
my $content = <$fh>;
close $fh;
```

Mais le code suivant est assez mauvais :

```
open my $fh, "foo" or die $!;
undef $/; # établit le mode slurp
my $content = <$fh>;
close $fh;
```

puisque un autre module peut vouloir lire des données d'un fichier quelconque dans le «mode ligne» par défaut; donc si le code que nous venons juste de présenter a été exécuté, la valeur globale de `$/` est maintenant changée pour tout autre code qui tourne à l'intérieur du même interpréteur Perl.

Habituellement, quand une variable est localisée, vous voulez être sûr que ce changement ait la plus petite étendue possible. Donc, à moins que vous ne soyez déjà à l'intérieur d'un bloc court `{}`, vous devriez en créer un vous-même. Par exemple:

```
my $content = '';
open my $fh, "foo" or die $!;
{
    local $/;
    $content = <$fh>;
}
close $fh;
```

Voici un exemple de la manière dont votre propre code peut aller de travers :

```
for (1..5){
    nasty_break();
    print "$_ ";
}
sub nasty_break {
    $_ = 5;
    # do something with $_
}
```

Vous vous attendez probablement à ce que code affiche :

```
1 2 3 4 5
```

mais à la place vous obtenez:

```
5 5 5 5 5
```

Pourquoi? Parce que nasty_break() modifie `$_` sans d'abord le localiser. La solution est d'ajouter local() :

```
local $_ = 5;
```

C'est facile de repérer le problème dans un exemple court comme celui-là, mais dans un code plus compliqué vous cherchez à avoir des problèmes si vous ne localisez pas les changements des variables spéciales.

Dans la liste suivante, on trouvera les variables scalaires d'abord, puis les tableaux, et enfin les tableaux associatifs.

$ARG

$_

La variable de stockage par défaut et l'espace de recherche de motif. Les paires suivantes sont équivalentes :

```
while (<>) {...}    # équivalent seulement dans while!
while (defined($_ = <>)) {...}

/^Subject:/
$_ =~ /^Subject:/

tr/a-z/A-Z/
$_ =~ tr/a-z/A-Z/

chomp
chomp($_)
```

Voici les endroits où Perl utilisera `$_` même si vous ne le précisez pas :

- Diverses fonctions unaires, notamment les fonctions comme `ord()` et `int()`, ainsi que tous les tests sur les fichiers (`-f`, `-d`) à l'exception de `-t`, qui utilise par défaut STDIN.
- Diverses fonctions de liste comme `print()` et `unlink()`.
- Les opérations de recherche de motif `m//`, `s///`, et `tr///` quand elles sont utilisées sans l'opérateur `=~`.
- La variable d'itération par défaut dans une boucle `foreach` si aucune autre variable n'est précisée.
- La variable implicite d'itération dans les fonctions `grep()` et `map()`.
- La variable par défaut où est stockée un enregistrement quand le résultat de `<FH>` s'autoteste et représente le critère unique d'un `while`. Attention : en dehors d'un test `while` cela ne marche pas.

(Moyen mnémonique: le *sous*-ligné est *sous*-entendu dans certaines opérations.)

$a

$b

Variables spéciales de paquetage quand sort() est utilisé, voyez sort in *perlfunc*. À cause de cette particularité $a et $b n'ont pas besoin d'être déclaré (en utilisant use vars, ou our()) même si vous utilisez le pragma vars strict. N'en faites pas des variables lexicales avec `my $a` ou `my $b` si vous voulez les utiliser dans un bloc ou une fonction de comparaison de sort().

$<*chiffres*>

Contient le groupement inclus dans les parenthèses correspondantes du dernier motif trouvé, sans prendre en compte les motifs trouvés dans les sous-blocs dont on est déjà sortis.(Mnémonique : comme \<*chiffres*>.) Ces variables sont en lecture seule et ont une portée dynamique étendue au BLOC courant.

$MATCH

$&

La chaîne de caractères trouvée par la dernière recherche de motif réussie (sans prendre en compte les motifs cachés dans un BLOC ou par un eval() inclus dans le BLOC courant). (Mnémonique : comme `&` dans certains éditeurs de texte.) Variable en lecture seule dont la portée dynamique s'étend au BLOC courant.

L'utilisation de cette variable, n'importe où dans un programme, pénalise considérablement les performances de toutes les recherches de motifs. Voir BUGS.

$PREMATCH

$‘

La chaîne de caractères qui précède tout ce qui a été trouvé au cours de la dernière recherche de motif réussie (non comptées les correspondances cachées dans un BLOC ou un `eval()` du BLOC courant). (Mnémonique : ‘ est souvent utilisé, en anglais, comme guillemet ouvrant dans les citations). Variable en lecture seule.

L'utilisation de cette variable, n'importe où dans un programme, pénalise considérablement les performances de toutes les recherches de motifs. Voir BUGS.

$POSTMATCH

$'

La chaîne de caractères qui suit tout ce qui a été trouvé au cours de la dernière recherche de motif réussie (non comptées les correspondances cachées dans un BLOC ou un eval() du BLOC courant). (Mnémonique : ' est souvent utilisé, en anglais, comme guillemet fermant dans les citations). Exemple :

```
local $_ = 'abcdefghi';
/def/;
print "$`:$&:$'\n";         # affiche abc:def:ghi
```

Variable en lecture seule dont la portée dynamique s'étend au BLOC courant.

L'utilisation de cette variable, n'importe où dans un programme, pénalise considérablement les performances de toutes les recherches de motifs. Voir BUGS.

$LAST_PAREN_MATCH

$+

Le texte du dernier groupement parenthésé trouvé par une recherche de motif. Utile si vous ignorez lequel des motifs d'une alternative a donné un résultat. Exemple :

```
/Version: (.*)|Revision: (.*)/ && ($rev = $+);
```

(Mnémonique: Soyez positif et allez de l'avant.) Variable en lecture seule dont la portée dynamique s'étend au BLOC courant.

$^N

Le texte du groupement parenthésé le plus récemment fermé (c.-à-d. le groupe avec la parenthèse fermée la plus à droite) du modèle de la dernière recherche de motifs réussie. (Mnémonique: la parenthèse eNchâssée (éventuelle) qui a été plus récemment a fermé.)

Essentiellement utilisé à l'intérieur de blocs `(?{...})` pour examiner le texte correspondant le plus récent. Par exemple, pour capturer efficacement le texte dans une variable (en plus de `$1`, `$2`, etc.), remplacez `(...)` par

```
(?:(...)(?{ $var = $^N }))
```

Fixer et utiliser `$var` de cette manière vous évite d'avoir à vous soucier de savoir exactement combien il y a d'ensemble de parenthèses.

Variable dont la portée dynamique s'étend au BLOC courant.

@LAST_MATCH_END

@+

Ce tableau contient les positions [offsets] des fins des sous-chaînes correspondant aux groupements de la dernière recherche de motifs réussie dans la portée dynamique courante. `$+[0]` est la position dans la chaîne de la fin de la recherche entière. C'est la même valeur que celle retournée par la fonction `pos` quand elle est appelée avec la variable sur laquelle porte la recherche. Le *n*-ième élément de ce tableau contient la position du *n*-ième groupement, donc `$+[1]` est la position où $1 fini, `$+[2]` la position où $2 fini, et ainsi de suite. Vous pouvez utiliser `$#+` pour déterminer combien il y a de groupements dans la dernière recherche réussie. Voyez les exemples donnés pour la variable `@-`.

$*

Fixé à une valeur entière non nulle pour rechercher dans une chaîne multilignes, à 0 (ou undef) pour signifier à Perl que la chaîne ne contient qu'une seule ligne, dans le but d'optimiser la recherche. La recherche de motif sur des chaînes contenant plusieurs retours lignes peut produire des résultats étranges quand "`$*`" est à 0 ou undef. La valeur par défaut est undef. (Mnémonique : * remplace plusieurs.) Cette variable n'influence que l'interprétation de "`^`" et de "`$`". Un retour ligne peut être recherché même si `$* == 0`.

L'utilisation de "`$*`" est obsolète dans les Perl modernes, et est supplantée par les modificateurs de recherche `/s` et `/m`.

Assigner une valeur non numérique à `$*` déclenche un avertissement (et fait que `$*` agit comme si `$* == 0`), tandis que lui donner une valeur numérique fait qu'un `int` implicite est appliqué à cette valeur.

HANDLE->input_line_number(EXPR)

$INPUT_LINE_NUMBER

$NR

$.

Le numéro de la ligne courante du dernier descripteur de fichier auquel vous avez accédé.

Chaque descripteur de fichier en Perl compte le nombre de lignes qui ont été lues par son intermédiaire. (Dépendant de la valeur de `$/`, l'idée de ce que Perl ce fait d'une ligne peut ne pas correspondre à la votre.) Quand une ligne est

lue d'un descripteur de fichier (via readline() or <>), ou quand tell() ou seek() est appelé sur lui, `$.` devient un alias du compteur de lignes pour ce descripteur de fichier.

Vous pouvez ajuster le compteur en assignant une valeur à `$.`, mais cela ne déplacera pas en fait le pointeur seek. *Localiser `$.`* ne localisera pas le compteur du descripteur de fichier. Au lieu de cela, il localisera la notion que Perl a du descripteur de fichier pour lequel `$.` est un alias.

`$.` est réinitialisé quand le descripteur de fichier est fermé, mais **non** quand un descripteur de fichier ouvert est réouvert sans avoir été fermé par close(). Comme <> ne provoque pas de fermeture explicite, le numéro de ligne augmente au travers des fichiers ARGV (voir les exemples dans eof in *perlfunc*).

Vous pouvez aussi utiliser `HANDLE->input_line_number(EXPR)` pour avoir accès au compteur de lignes d'un descripteur de fichier donné sans avoir à vous tracasser de savoir quel est le dernier descripteur utilisé.

(Mnémonique : beaucoup de programmes utilisent "." pour *point*er la ligne en cours)

IO::Handle->input_record_separator(EXPR)

$INPUT_RECORD_SEPARATOR

$RS

$/

Le séparateur d'enregistrement en lecture, par défaut retour-ligne. Il influence l'idée que Perl se fait d'une "ligne". Fonctionne comme la variable RS de **awk**, y compris le traitement des lignes vides comme des délimiteurs si initialisé à vide. (Note : une ligne vide ne peut contenir ni espace, ni tabulation.) Vous pouvez le définir comme une chaîne de caractères pour correspondre à un délimiteur multicaractère, ou à `undef` pour lire jusqu'à la fin du fichier. Notez que le fixer à `"\n\n"` est légèrement différent que de le fixer à `""` si le fichier contient plusieurs lignes vides consécutives. Le positionner à `""` traitera deux lignes vides consécutives (ou plus) comme une seule. Le positionner à `"\n\n"` implique que le prochain caractère lu appartient systématiquement à un nouveau paragraphe, même si c'est un retour-ligne. (Mnémonique: / est utilisé comme séparateur de ligne quand on cite une poésie.)

```
local $/;              # activer le mode "slurp"
local $_ = <FH>;       # Fichier complet depuis la position courante
s/\n[ \t]+/ /g;
```

Attention : La valeur de `$/` doit être du texte et non une expression régulière. Il faut bien laisser quelque chose à **awk** :-)

Initialiser `$/` avec une référence à un entier, un scalaire contenant un entier, ou un scalaire convertissable en entier va provoquer la lecture d'enregistrements au lieu de lignes, avec une taille maximum par enregistrement correspondant à l'entier en référence. Donc :

```
local $/ = \32768;              # ou \"32768", ou \$var_contenant_32768
open my $fh, $myfile or die $!;
local $_ = <$fh>;
```

va lire un enregistrement d'une longueur maximale de 32768 octets depuis FILE. Si votre fichier ne contient pas d'enregistrements (ou si votre système d'exploitation ne supporte pas les fichiers d'enregistrements), vous obtiendrez probablement des valeurs incohérentes à chaque lecture. Si l'enregistrement est plus long que la taille spécifiée, il vous faudra le lire en plusieurs fois.

Sur VMS, les lectures d'enregistrements sont faites avec l'équivalent de `sysread`, donc il vaut mieux éviter de mélanger les lectures en mode enregistrements et en mode lignes sur le même fichier. (Cela n'est généralement pas un problème, puisque les fichiers que vous souhaiteriez lire en mode enregistrement sont probablement illisibles en mode ligne). Les systèmes non VMS effectuent des Entrées/Sorties standards, donc il est possible de mélanger les deux modes de lecture.

Voir aussi Les retours chariots in *perlport* et `$.`.

HANDLE->autoflush(EXPR)

$OUTPUT_AUTOFLUSH

$|

Si initialisé à une valeur différente de zéro, force une actualisation immédiatement et juste après chaque opération de lecture/écriture sur le canal de sortie sélectionné courant. La valeur par défaut est 0 (que le canal de sortie soit bufferisé par le système ou non; `$|` vous indique seulement si vous avez explicitement demandé à Perl d'actualiser après chaque écriture). Notez que STDOUT est typiquement bufferisé par ligne en sortie écran et par blocs sinon. Initialiser cette variable est surtout utile dans le cas d'une redirection de sortie, par exemple si vous exécutez un script Perl avec **rsh** et que vous voulez voir le résultat au fur et à mesure. Cela n'a aucun effet sur les buffers d'entrée. Voir getc in *perlfunc* pour cela. (Mnémonique : l'édition actualisée de vos redirections |)

IO::Handle->output_field_separator EXPR

$OUTPUT_FIELD_SEPARATOR

$OFS

$,

Le séparateur de champs pour l'opérateur print. Si définie, cette valeur est affichée entre chacun des arguments de print. La valeur par défaut est `undef`. (Mnémonique : Ce qui est imprimé quand vous avez une "," dans vos champs)

IO::Handle->output_record_separator EXPR

$OUTPUT_RECORD_SEPARATOR

$ORS

$

Le séparateur d'enregistrements pour l'opérateur print. Si définie, cette valeur est affichée après le dernier argument d'un print. (Mnémonique : positionnez "`$\`" au lieu d'ajouter \n à la fin de chaque impression. Ou : Comme `$/`, mais c'est ce que vous "obtenez" de Perl.) (NdT. get **back** = "obtenez", \ = **back**slash)

$LIST_SEPARATOR

$"

Même chose que "`$,`", sauf que cela s'applique aux tableaux de valeurs interpolées dans une chaîne de caractères entre guillemets doubles (ou toute chaîne interprétée d'une manière équivalente). La valeur par défaut est "espace". (Mnémonique : évident, je pense).

$SUBSCRIPT_SEPARATOR

$SUBSEP

$;

Le séparateur d'indices pour l'émulation de tableaux à plusieurs dimensions. Si vous vous référez à un élément de type tableau associatif comme

```
$foo{$a,$b,$c}
```

Cela signifie en fait

```
$foo{join($;, $a, $b, $c)}
```

Mais n'utilisez pas

```
@foo{$a,$b,$c}      # une tranche -- notez le @
```

qui signifie

```
($foo{$a},$foo{$b},$foo{$c})
```

La valeur par défaut est "\034", la même que pour le SUBSEP en **awk**. Si vos clefs contiennent des valeurs binaires, il se peut qu'il n'y ait aucune valeur sûre pour "`$;`". (Mnémonique : la virgule (le séparateur d'indices) est un demi-point-virgule. Ouais, je sais; c'est tiré par les cheveux, mais "`$,`" est déjà utilisé pour quelque chose de plus important.)

Envisagez d'utiliser de "vrais" tableaux à plusieurs dimensions comme décrit dans *perllol*.

$#

C'est le format de sortie des nombres. Cette variable est une pâle tentative d'émulation de la variable OFMT de **awk**. Il y a des cas, cependant, où Perl et **awk** ont des vues différentes sur la notion de numérique. La valeur par défaut est *%.n*g, où *n* est la valeur de la macro DBL_DIG du *float.h* de votre système. C'est différent de la valeur par défaut de OFMT en **awk** qui est "%.6g", donc il vous faut initialiser `$#` explicitement pour obtenir la valeur **awk**. (Mnémonique : # est le signe des nombres – numéros)

L'utilisation de "`$#`" est obsolète.

HANDLE->format_page_number(EXPR)

$FORMAT_PAGE_NUMBER

$%

Le numéro de la page en cours sur le canal de sortie en cours. Utilisés avec les formats. (Mnémonique : % est le numéro de page pour **nroff**.)

HANDLE->format_lines_per_page(EXPR)

$FORMAT_LINES_PER_PAGE

$=

La longueur de la page courante (en nombre de lignes imprimables) du canal de sortie en cours. Le défaut est 60. (Mnémonique : = est formé de lignes horizontales.)

HANDLE->format_lines_left(EXPR)

$FORMAT_LINES_LEFT

$-

Le nombre de lignes restantes sur la page en cours du canal de sortie courant. Utilisé avec les formats. (Mnémonique : lignes_sur_la_page - lignes_imprimées.)

@LAST_MATCH_START

@-

`$-[0]` est la position [offset] du début de la dernière recherche de motif réussie. `$-[`*n*`]` est la position du début de la sous-chaîne qui correspond au *n*-ième groupement, ou undef si le groupement ne correspond pas.

Donc, après une recherche de motif dans $_, $& coïncide avec `substr $_, $-[0], $+[0] - $-[0]`. De la même manière, $*n* coïncide avec `substr $_, $-[n], $+[n] - $-[n]` si `$-[n]` est défini, et $+ coïncide avec `substr $_, $-[$#-], $+[$#-]`. On peut utiliser `$#-` pour trouver le dernier groupement dans la dernière recherche réussie. En opposition avec `$#+`, le nombre de groupements dans l'expression régulière. Comparez avec `@+`.

Ce tableau contient les positions des débuts des sous-chaînes correspondant aux groupements de la dernière recherche de motif réussie dans la portée dynamique courante. `$-[0]` est la position dans la chaîne du début de la chaîne trouvée entière. Le *n*-ième élément de ce tableau contient la position du *n*-ième groupement, donc `$+[1]` est la position où $1 commence, `$+[2]` est la position où $2 commence, et ainsi de suite.

Après une recherche sur une variable $var :

- `$`` est semblable à `substr($var, 0, $-[0])`
- `$&` est semblable à `substr($var, $-[0], $+[0] - $-[0])`
- `$'` est semblable à `substr($var, $+[0])`
- `$1` est semblable à `substr($var, $-[1], $+[1] - $-[1])`
- `$2` est semblable à `substr($var, $-[2], $+[2] - $-[2])`
- `$3` est semblable à `substr($var, $-[3], $+[3] - $-[3])`

HANDLE->format_name(EXPR)

$FORMAT_NAME

$~

Le nom du format de rapport courant pour le canal de sortie sélectionné. La valeur par défaut est le nom du descripteur de fichier. (Mnémonique : frère de "`$^`".)

HANDLE->format_top_name(EXPR)

$FORMAT_TOP_NAME

$^

Nom du format de l'en-tête de page courant pour le canal de sortie sélectionné. La valeur par défaut est le nom du descripteur de fichier suivi de _TOP. (Mnémonique : pointe vers le haut de la page.)

IO::Handle->format_line_break_characters EXPR

$FORMAT_LINE_BREAK_CHARACTERS

$:

L'ensemble des caractères courants après lesquels une chaîne de caractères peut être coupée pour passer à la ligne (commençant par ^) dans une sortie formatée. La valeur par défaut est " \n-", pour couper sur les espaces ou les traits d'unions. (Mnémonique : en poésie un "deux-points" fait partie de la ligne.)

IO::Handle->format_formfeed EXPR

$FORMAT_FORMFEED

$^L

Ce que les formats utilisent pour passer à la page suivante. La valeur par défaut est "\f".

$ACCUMULATOR

$^A

La valeur courante de la pile de la fonction write() pour formater les lignes avec `format()`. Un format contient des commandes formline(), qui stockent leurs résultats dans `$^A`. Après appel à son format, write() sort le contenu de `$^A` et le réinitialise. Donc vous ne voyez jamais le contenu de `$^A`, à moins que vous n'appeliez `formline()` vous-même, et ne regardiez le contenu. Voir *perlform* et formline() in *perlfunc* .

$CHILD_ERROR

$?

Le statut retourné par la dernière fermeture d'une redirection, une commande 'anti-apostrophe' (`), un appel réussi à `wait()` ou `waitpid()`, ou un opérateur system(). Notez que c'est le statut retourné par l'appel système wait() (ou cela lui ressemble). Le code de terminaison du sous-processus est (`$? >> 8`), et `$? & 127` indique quel signal (s'il y a lieu) a arrêté le processus, enfin `$? & 128` indique s'il y a eu un vidage mémoire [core dump]. (Mnémonique : similaire à **sh** et **ksh**.)

De plus, si la variable `h_errno` est supportée en C, sa valeur est retournée par $? si n'importe laquelle des fonctions `gethost*()` échoue.

Si vous avez installé un gestionnaire de signal pour `SIGHLD`, la valeur `$?` sera la plupart du temps erronée en dehors de ce gestionnaire.

A l'intérieur d'un sous-programme `END`, `$?` contient la valeur qui sera passée à `exit()`. Vous pouvez modifier `$?` dans un sous-programme `END` pour changer le code de sortie d'un script. Par exemple :

```
END {
    $? = 1 if $? == 255;  # die lui donnerait la valeur 255
}
```

Sous VMS, la déclaration `use vmsish 'status'` fait renvoyer à `$?` le code de sortie réel de VMS, plutôt que le code d'émulation POSIX habituel status; voir **$?** in *perlvms* pour les details.

Voir aussi Indicateurs d'erreur.

${^ENCODING}

La *référence d'objet* à l'objet Encode qui est utilisé pour convertir le code source en Unicode. Grâce à cette variable, votre script Perl n'a pas besoin d'être écrit en UTF-8. La valeur par défaut est *undef*. La manipulation directe de cette variable est fortement déconseillée. Voyez *encoding* pour plus de détails.

$OS_ERROR

$ERRNO

$!

Dans un contexte numérique, contient la valeur courante de la variable `errno` ou, en d'autres termes, si un appel système ou à une bibliothèque échoue, il fixe cette variable. Cela signifie que la valeur de `$!` n'est significative qu'*immédiatement* après un **échec**.

```
if (open(FH, $filename)) {
    # Ici $! est sans signification.
    ...
} else {
    # Ici SEULEMENT $! est significative.
    ...
    # Ici, à nouveau, $! peut être sans signification.
}
# Puisqu'ici nous pouvons avoir ou succès ou échec,
# $! est sans signification ici.
```

Ci-dessus, *sans signification* est mis pour n'importe quoi : zéro, non-zéro, `undef`. Un appel réussi au système ou a une bibliothèque ne met pas la variable à zéro.

Dans un contexte textuel, contient le texte de l'erreur. Vous pouvez affecter un nmobre à `$!` pour fixer `errno` si, par exemple, vous voulez utiliser `"$!"` pour retourner le texte de l'erreur *n*, ou si vous voulez fixer la valeur de l'opérateur `die()`. (Mnémonique : Plantage !)

Voir aussi Indicateurs d'erreur.

%!

Chaque élément de `%!` à une valeur vraie seulement si `$!` est fixé à cette valeur. Par exemple, `$!{ENOENT}` est vrai si et seulement si la valeur courante de `$!` est `ENOENT`; c'est-à-dire si la plus récente erreur a été "No such file or directory" (ou son équivalent moral : tous les systèmes d'exploitation ne donne pas exactement cette erreur et encore moins tous les langages). Pour vérifier si une clé particulière est significative sur votre système utilisez `exists $!{the_key}`; pour une liste des clés légales, utilisez `keys %!`. Voir *Errno* pour plus d'information, et voir aussi ci-dessus pour la validité de `$!`.

$EXTENDED_OS_ERROR

$^E

Information d'erreur spécifique au système d'exploitation. Actuellement diffère de `$!` seulement sous VMS, OS/2 et Win32 (et MacPerl). Sur toutes les autres plates-formes, `$^E` est équivalent à `$!`.

Sous VMS, `$^E` fournit la valeur du statut VMS de la dernière erreur système. Les informations sont plus spécifiques que celles fournies par `$!`. Ceci est particulièrement important quand `$!` est fixé à **EVMSERR**.

Sous OS/2, `$^E` correspond au code d'erreur du dernier appel API, soit au travers de CRT, soit directement depuis Perl.

Sous Win32, `$^E` retourne toujours l'information relative au dernier appel Win32 `GetLastError()`, qui décrit le dernier code d'erreur de l'API Win32. La plupart des applications spécifiques Win32 reportent les erreurs via `$^E`. ANSI C et UNIX positionnent `errno`, donc les programmes Perl les plus portables utiliseront `$!` pour remonter les messages d'erreur.

Les avertissements mentionnés dans la description de `$!` s'appliquent généralement à `$^E`. (Mnémonique : Extra-Explication d'Erreur)

Voir aussi Indicateurs d'erreur.

$EVAL_ERROR

$@

Le message d'erreur de syntaxe de la dernière commande `eval()`. Si $@ est la chaîne nulle, le dernier eval() s'est exécuté correctement (bien que l'opération à exécuter ait pu échouer en mode normal). (Mnémonique : @ttention à l'erreur de syntaxe !)

Les messages d'avertissements ne sont pas récupérés dans cette variable. Vous pouvez toutefois construire une routine de traitement des avertissements en positionnant `$SIG{__WARN__}` comme décrit plus loin.

Voir aussi Indicateurs d'erreur.

$PROCESS_ID

$PID

$$

Numéro du processus Perl exécutant ce script. Vous devriez considérer cette variable comme étant en lecture seule, bien qu'elle puisse être modifiée à travers des appels à `fork()`. (Mnémonique : comme en shell.)

Note pour les utilisateurs de Linux : sur Linux, les fonctions C `getpid()` et `getppid()` retourne des valeurs différentes pour des fils d'exécution (threads) différents. Afin d'être portable, ce comportement n'est pas celui de `$$` dont la valeur reste cohérente entre les fils d'exécution. Si vous souhaitez appeler la fonction `getpid()` réelle, vous pouvez utiliser le module CPAN `Linux::Pid`.

$REAL_USER_ID

$UID

$<

L'uid (id utilisateur) réel du processus. (Mnémonique : l'uid d'*OU* vous venez si vous exécutez setuid.) Vous pouvez changer l'uid réel et l'uid effectif en même temps en utilisant POSIX::setuid(). Puisque une modification de $< nécessite un appel système, vérifiez $! après chaque tentative de modification pour détecter une éventuelle erreur.

$EFFECTIVE_USER_ID

$EUID

$>

L'uid effectif de ce processus. Exemple :

```
$< = $>;            # Positionne l'uid réel à la valeur de l'iud effectif
($<,$>) = ($>,$<);  # Inverse les uid réel et effectif
```

Vous pouvez à la fois changer l'uid réel et l'uid effectif en même temps en utilisant POSIX::setuid().

(Mnémonique : l'uid *VERS* lequel vous alliez, si vous exécutez setuid.) Note : `$<` and `$>` peuvent être inversés seulement sur les machines supportant `setreuid()`. Puisque une modification de $> nécessite un appel système, vérifiez $! après chaque tentative de modification pour détecter une éventuelle erreur.

$REAL_GROUP_ID

$GID

$(

Le gid (id de groupe) réel du processus. Si vous êtes sur une machine qui supporte l'appartenance simultanée à plusieurs groupes, renvoie une liste des groupes auxquels vous appartenez, séparés par des espaces. Le premier nombre retourné est le même que celui retourné par `getgid()`, les autres sont ceux retournés par `getgroups()`, avec possibilité de doublon entre l'un d'eux et le premier nombre.

Toutefois une valeur assignée à `$(` doit être un nombre unique utilisé pour fixer le gid réel. Donc la valeur donnée par `$(` ne doit *pas* être réassignée à `$(` sans être forcée en numérique, par exemple en lui ajoutant 0.

Vous pouvez à la fois changer le gid réel et le gid effectif en même temps en utilisant POSIX::setgid(). Puisque une modification de $(nécessite un appel système, vérifiez $! après chaque tentative de modification pour détecter une éventuelle erreur.

(Mnémonique : les parenthèses sont utilisées pour *GROUPER* les choses. Le gid réel est le groupe que vous laissez à votre *GAUCHE*, si vous utilisez setgid.) (NdT: *GAUCHE* = parenthèse gauche)

$EFFECTIVE_GROUP_ID

$EGID

$)

Le gid effectif du processus. Si vous êtes sur une machine qui supporte l'appartenance simultanée à plusieurs groupes, renvoie une liste des groupes auxquels vous appartenez, séparés par des espaces. Le premier nombre retourné est le même que celui retourné par `getegid()`, les autres sont ceux retournés par `getgroups()`, avec possibilité de doublon entre l'un d'eux et le premier nombre.

De la même façon, une valeur assignée à `$)` doit être une liste de nombres séparés par des espaces. Le premier nombre est utilisé pour fixer le gid effectif, et les autres (si présents) sont passés à `setgroups()`. Pour obtenir le résultat d'une liste vide pour `setgroups()`, il suffit de répéter le nouveau gid effectif; c'est à dire pour forcer un gid effectif de 5 et un setgroups() vide il faut utiliser : `$) = "5 5"` .

Vous pouvez à la fois changer le gid réel et le gid effectif en même temps en utilisant POSIX::setgid() (utilise seulement un unique argument numérique). Puisque une modification de $) nécessite un appel système, vérifiez $! après chaque tentative de modification pour détecter une éventuelle erreur.

(Mnémonique : les parenthèses sont utilisées pour *GROUPER* les choses. Le gid effectif est le groupe qui vous donne le bon *DROIT* pour vous si vous exécutez setgid.) (NdT: *DROIT* = parenthèse droite)

Note : `$<`, `$>`, `$(` et `$)` peuvent être positionnés seulement sur les machines qui supportent les fonctions correspondantes *set[re][ug]id()*. `$(` et `$)` peuvent être inversée seulement sur les machines supportant `setregid()`.

$PROGRAM_NAME

$0

Contient le nom du script Perl en cours d'exécution.

Sur certains systèmes (pas tous), assigner une valeur à "`$0`" modifie la zone d'argument que le programme `ps` voit. Sur certaines plateformes vous devez une option spéciale de `ps` ou un `ps` différent pour voir cette modification. C'est utile plutôt pour indiquer l'état courant du programme que pour cacher le programme en cours d'exécution. (Mnémonique : comme en **sh** et **ksh**.)

Notez qu'il existe une limite spécifique à chaque plateforme pour la longueur maximale de `$0`. Dans les cas les plus extrêmes, cette limite est la longueur de la valeur initiale de `$0`.

Sur certaines plateformes, il y a un remplissage arbitraire par, par exemple, des espaces après le nom modifié tel que vu par `ps`. Sur certaines plateformes, ce remplissage s'étend à la longueur initiale du nom quoique vous fassiez (c'est le cas de Linux 2.2 par exemple).

Note pour les utilisateurs de BSD : fixer la valeur de `$0` ne retire pas complètement "perl" de la sortie de **ps(1)**. Par exemple, donner la valeur "`foobar`" à `$0` peut donner le résultat "`perl: foobar (perl)`" (la présence ou l'absence du préfixe "`perl:`" ou du suffixe "`(perl)`" dépend de la variante et de la version BSD utilisée). C'est une caractéristique du système d'exploitation à laquelle Perl ne peut rien.

Dans des scripts multi-fils (multi-threads), chaque fil peut modifier sa copie de `$0` et Perl les coordonne afin que ce changement soit visible par ps(1) (en supposant que le système d'exploitation le permette). Notez que la valeur `$0` des autres fils n'est pas modifiée puisqu'ils enont une copie qui leur est propre.

$[

L'index du premier élément dans un tableau, et du premier caractère dans une sous-chaîne de caractère. La valeur par défaut est 0, mais vous pouvez la fixer à 1 pour faire ressembler Perl à **awk** (ou Fortran) quand vous utilisez les index ou les fonctions index() et substr(). (Mnémonique : [commence les index)

Depuis Perl 5, fixer la valeur de "`$[`" est traité comme une directive de compilation et ne peut influencer le comportement des autres fichiers (ce qui explique que vous ne puissiez lui affecter qu'une valeurs constante à la compilation). Son usage est fortement déconseillé.

Notez que, contrairement à d'autres directives de compilation (tel que *strict*), la modification de `$[` est perçue dans tout le fichier, et donc en dehors de sa portée lexicale normale. En revanche, vous pouvez utiliser local() pour limiter sa portée à un bloc lexical.

$]

La version + le niveau de patch / 1000 de l'interpréteur Perl. Cette variable peut être utilisée pour déterminer si l'interpréteur Perl exécutant un script est au niveau de version souhaité. (Mnémonique : Cette version du Perl est-elle accrochée droite) (Ndt : Les jeux de mots utilisés dans les mnémoniques anglais sont des plus délicats à rendre en français...) Exemple :

```
warn "No checksumming!\n" if $] < 3.019;
```

Voir également la documentation de `use VERSION` et `require VERSION` pour un moyen pratique d'empêcher l'exécution d'un script si l'interpréteur est trop vieux.

Lorsque vous testez cette variable, pour éviter les problèmes dus aux imprécisions numériques, utilisez les tests d'inégalité < et > plutôt que les tests contenant l'égalité comme <=, == et >=.

La représentation numérique en point flottant peut quelquefois amener des comparaisons numériques inexactes. Voir `$^V` pour une représentation plus moderne du numéro de la version Perl qui permet des comparaisons exactes sur les chaînes.

$COMPILING

$^C

La valeur courante de l'indicateur binaire [flag] associé au commutateur **-c**. Principalement utilisé avec **-MO=...** pour permettre au code de modifier son comportement pendant qu'il est compilé, comme par exemple pour utiliser AUTOLOAD pendant la compilation plutôt que le chargement différé normal. Voir See *perlcc*. Fixer `$^C = 1` est équivalent à appeler `B::minus_c`.

$DEBUGGING

$^D

La valeur des options de débuggage. (Mnémonique : valeur de l'option **-D**.) Peut être lue ou modifiée. Comme pour l'option équivalente en ligne de commande, vous pouvez utiliser soit une valeur numérique (`$^D = 10`) soit une valeur symbolique (`$^D = "st"`).

$SYSTEM_FD_MAX

$^F

Le nombre maximum de descripteurs de fichier système, habituellement 2. Les descripteurs de fichiers système sont passés aux processus lancés par exec(), alors que ce n'est pas le cas pour les autres descripteurs de fichiers. De plus, pendant un open(); les descripteurs de fichiers système sont préservés même si open() échoue. (Les autres descripteurs sont fermés avant qu'un open() ne soient tenté.). Le statut fermeture-sur-exec d'un descripteur de fichier sera décidé suivant la valeur de `$^F` au moment de l'ouverture du fichier, pipe ou socket correspondant et non au moment de l'exec().

$^H

ATTENTION : Cette variable est disponible pour utilisation interne uniquement. Sa disponibilité, son comportement et son contenu sont soumis à changement sans avis.

Cette variable contient des directives de compilation pour l'interprète Perl. À la fin de la compilation d'un BLOC la valeur de cette variable est restaurée à la valeur qu'elle avait avant que l'interprète ne commence la compilation du BLOC.

Quand perl commence à analyser toute construction d'un bloc qui fournit une portée lexicale (par exemple, corps d'un eval, fichier requis, corps d'un sous-programme, corps de boucle, ou bloc conditionnel), la valeur existante de $^H est sauvegardée, mais sa valeur est inchangée. Quand la compilation du bloc est terminée, elle reprend la valeur sauvegardée. Entre les points où sa valeur est sauvegardée et où elle est restaurée, le code des blocs BEGIN est libre de changer la valeur de $^H.

Ce comportement fournit la sémantique du traitement de portée lexicale, et est utilisé, par exemple, dans le pragma `use strict`.

Le contenu devrait être un nombre entier; les différents bits de celui-ci sont utilisés comme indicateurs binaires de pragma. Voici un exemple:

```
sub add_100 { $^H |= 0x100 }

sub foo {
    BEGIN { add_100() }
    bar->baz($boon);
}
```

Considérons ce qui se passe pendant l'exécution du bloc BEGIN. À ce moment, le bloc BEGIN a déjà été compilé, mais le corps de foo() est encore à compiler. Par conséquent, la nouvelle valeur de $^H ne sera visible seulement que pendant que le corps de foo() est compilé.

La substitution du bloc BEGIN précédent par :

```
BEGIN { require strict; strict->import('vars') }
```

montre comment `use strict 'vars'` est implémenté. Voici une version conditionnelle du même pragma lexical :

```
BEGIN { require strict; strict->import('vars') if $condition }
```

%^H

ATTENTION : Cette variable est disponible pour utilisation interne uniquement. Sa disponibilité, son comportement et son contenu sont soumis à changement sans avis.

Le hachage %^H fournit la même sémantique du traitement de portée lexicale que $^H. Ceci le rend utile pour implémenter des pragmas de portée lexicale.

$INPLACE_EDIT

$^I

La valeur courante de l'extension "édition sur place". Utilise `undef` pour mettre hors service l'édition sur place. (Mnémonique : valeur de l'option **-i**.)

$^M

Par défaut, le dépassement de mémoire ne peut pas être capturé et entraîne une erreur fatale. Cependant, s'il est compilé pour cela, Perl peut utiliser le contenu de `$^M` comme une réserve d'urgence après un die() dû à un dépassement de mémoire. Supposons que votre Perl ait été compilé avec l'option `-DPERL_EMERGENCY_SBRK` et utilise le malloc de Perl. Dans ce cas

```
$^M = 'a' x (1<<16);
```

allouera un buffer de 64K à utiliser en cas d'urgence. Voir le fichier *INSTALL* de votre distribution pour savoir comment ajouter vos propres options C de compilation lors de la compilation de perl. Pour décourager l'utilisation abusive de cette fonction avancée, il n'y a pas de noms longs English pour cette variable.

$OSNAME

$^O

Le nom du système d'exploitation utilisé pour compiler cette copie de Perl, déterminé pendant le processus de configuration. Cette valeur est identique à `$Config{'osname'}`. Voir aussi *Config* et le commutateur en ligne de commande **-V** documenté dans *perlrun*.

Sur les plate-formes Windows $^O n'est pas très utile: puisqu'elle donne toujours `MSWin32`, elle ne fait pas la différence entre 95/98/ME/NT/2000/XP/CE / .NET. Utilisez Win32::GetOSName() ou Win32::GetOSVersion() (voir *Win32* et *perlport*) pour distinguer ces variantes.

${^OPEN}

Une variable interne utilisée par PerlIO. Une chaîne en deux parties, séparées par un octet `\0`, la première partie décrit les couches d'entrée, la seconde partie décrit les couches de sortie.

$PERLDB

$^P

Variable interne pour le débuggage. La signification des différents bits est sujet à changement, mais est actuellement :

1. x01
   Débuggage d'un sous-programme enter/exit.
2. x02
   Débuggage ligne à ligne.
3. x04
   Annuler les options d'optimisation.
4. x08
   Préserver plus de données pour les inspections interactives à venir.
5. x10
   Garder des informations sur les lignes sources où un sous-programme est défini.
6. x20
   Démarrer en mode pas à pas.
7. x40
   Dans le rapport, utiliser l'adresse du sous-programme au lieu de nom.
8. x80
   Mettre aussi les `goto &subroutine` dans le rapport.
9. x100
   Fournir des noms de fichier significatifs pour les evals, basés sur l'endroit où ils ont été compilés.
10. x200
    Fournir des noms significatifs pour les sous-programmes anonymes, basés sur l'endroit où ils ont été compilés.

11. x400
    Assertion de débogage de l'entrée/sortie des sous-programmes.

Note : Certains bits peuvent avoir un sens uniquement à la compilation , d'autres seulement à l'exécution. C'est un mécanisme nouveau, et les détails peuvent varier.

$LAST_REGEXP_CODE_RESULT

$^R

Résultat de l'évaluation de la dernière utilisation réussie d'une expression régulière `(?{ code })`. Voir *perlre*. Cette variable est en lecture/écriture.

$EXCEPTIONS_BEING_CAUGHT

$^S

Étant courant de l'interpréteur.

```
$^S         État
---------   -------------------
undef       Analyse d'un module/eval
true (1)    En cours d'exécution d'un eval
false (0)   Autre
```

Le premier état peut appraître dans un gestionnaire (handler) $SIG{__DIE__} ou $SIG{__WARN__}.

$BASETIME

$^T

Heure à laquelle le script a démarré, en secondes depuis le 01/01/1970. Les valeurs retournées par les tests de fichiers **-M**, **-A**, et **-C** sont basées sur cette valeur.

${^TAINT}

Indique si le mode souillé est actif ou non. 1 si actif (le programme a été lancé avec l'option **-T**), 0 si inactif, -1 si seuls les avertissements sont activés (avec **-t** ou **-TU**).

${^UNICODE}

Indique l'état de certains réglage Unicode de Perl. Voir la documentation de l'option `-C` dans *perlrun* pour plus d'information sur les valeurs possibles. Cette variable est positionnée au lancement de Perl et n'est plus modifiable ensuite.

${^UTF8LOCALE}

Cette variable indique si un locale UTF-8 a été détecté par perl au lancement. Cette information est utilisée par perl lorsqu'il est dans le mode d'ajustement de l'utf-8 au locale (lorsqu'on utilise l'option `-CL`) ; voir *perlrun* pour plus d'informations.

$PERL_VERSION

$^V

Les numéros de révision, version et sous-version de l'interpréteur Perl, sous la forme d'une chaîne composée des caractères portant ces numéros. Ainsi, dans Perl v5.6.0, elle est égale à `chr(5) . chr(6) . chr(0)` et `$^V eq v5.6.0` retourne vrai. Notez que les caractères dans cette chaîne peuvent être potentiellement des caractères Unicode.

On peut l'utiliser pour déterminer si l'interpréteur Perl qui exécute un script est dans la bonne gamme de versions. (Mnémonique: utilisez ^V pour contrôle de la Version.) Exemple:

```
warn "No \"our\" declarations!\n" if $^V and $^V lt v5.6.0;
```

Pour convertir `$^V` en sa représentation en chaîne de caractères, utilisez le motif `"%vd"` de sprintf() :

```
printf "version is v%vd\n", $^V;  # Version de Perl
```

Voir la documentation de `use VERSION` et de `require VERSION` pour une manière commode de terminer le script si l'interpréteur Perl courant est trop ancien.

Voyez aussi `$]` pour une plus ancienne représentation de la version Perl.

$WARNING

$^W

Valeur de l'option 'warning' (avertissement), initialement vrai si **-w** est utilisé, faux sinon, mais peut être directement modifié. (Mnémonique : comme l'option **-w**.) Voir aussi *warnings*.

${^WARNING_BITS}

L'ensemble des contrôles courants demandés avec le pragma `use warnings` Voir la documentation de `warnings` pour plus de détails.

$EXECUTABLE_NAME

$^X

Le nom utilisé pour exécuter la copie courante de Perl, vient de `argv[0]` en C ou de */proc/self/exe* (lorsque ça existe).

Selon le système d'exploitation hôte, la valeur de $^X peut être un chemin relatif ou absolu du fichier programme perl, ou peut être la chaîne utilisée pour invoquer perl et non le chemin du programme perl. Également, la plupart des systèmes d'exploitation permettent l'invocation de programmes qui ne sont pas dans la variable d'environnement PATH, donc il n'y a aucune garantie que la valeur de $^X soit dans PATH. Pour VMS, la valeur peut ou peut ne pas inclure de numéro de version.

Vous pouvez généralement utiliser la valeur de $^X pour réinvoquer une copie indépendante du même perl qui est en train de tourner, par exemple :

```
@first_run = `$^X -le "print int rand 100 for 1..100"`;
```

Mais rappelez-vous que tous les systèmes d'exploitation ne supportent pas le forking ou la capture de sortie des commandes, donc cette instruction complexe peut ne pas être portable.

Il n'est pas sans danger d'utiliser la valeur de $^X comme un nom de chemin vers un fichier car certains systèmes d'exploitation qui ont un suffixe obligatoire pour les fichiers exécutables n'exige pas l'usage du suffixe quand on invoque une commande. Pour convertir la valeur de $^X en un nom de chemin, utilisez les instructions suivantes:

```
# Construit un ensemble de noms de fichier (pas des noms de commande).
use Config;
$this_perl = $^X;
if ($^O ne 'VMS')
   {$this_perl .= $Config{_exe}
        unless $this_perl =~ m/$Config{_exe}$/i;}
```

Parce que beaucoup de systèmes d'exploitation permettent à n'importe qui possédant un accès en lecture au fichier programme Perl d'en faire une copie, de modifier la copie, et ensuite d'exécuter cette copie, le programmeur Perl soucieux de sécurité doit prendre soin d'invoquer la copie installée de perl et non pas la copie référencée par $^X. Les instructions suivantes permettent d'atteindre ce but, et produisent un chemin qui peut être invoqué comme une commande ou qui peut être référencé comme un fichier.

```
use Config;
$secure_perl_path = $Config{perlpath};
if ($^O ne 'VMS')
   {$secure_perl_path .= $Config{_exe}
        unless $secure_perl_path =~ m/$Config{_exe}$/i;}
```

ARGV

Le descripteur de fichier spécial qui itère sur les noms de fichier de la ligne de commande de `@ARGV`. Habituellement écrit comme le descripteur de fichier nul dans l'opérateur angle <>. Notez qu'actuellement `ARGV` n'a son effet magique qu'avec seulement l'opérateur <>; ailleurs c'est juste un descripteur de fichier ordinaire qui correspond au dernier fichier ouvert par <>. En particulier, passer `\*ARGV` comme paramètre à une fonction qui attend un descripteur de fichier ne permet pas à cette fonction de lire automatiquement le contenu de tous les fichiers de `@ARGV`.

$ARGV

Contient le nom du fichier courant quand on lit depuis <>.

@ARGV

Contient les arguments de la ligne de commande du script. `$#ARGV` est généralement le nombre d'argument moins 1 car `$ARGV[0]` est le premier argument, et *non pas* le nom du script qui est dans `"$0"`. Voir `"$0"` pour le nom du script.

ARGVOUT

Le descripteur de fichier spécial qui pointe le fichier de sortie ouvert courant quand on fait de l'édition sur place avec **I**. Utile quand vous devez faire beaucoup d'insertions et que vous ne voulez pas modifier $_. Voir *perlrun* pour le commutateur **I**.

@F

Le tableau @F contient les champs de chaque ligne lue quand le mode auto-découpage (autosplit) est activé. Voir *perlrun* pour le commutateur **A**. Ce tableau est spécifique au paquetage et doit être déclaré ou doit être invoqué avec un nom pleinement qualifié s'il n'est pas dans le paquetage main quand on utilise `strict 'vars'`.

@INC

Contient la liste des répertoires où chercher pour évaluer les constructeurs `do EXPR`, `require`, ou `use`. Il est constitué au départ des arguments **-I** de la ligne de commande, suivi du chemin par défaut de la bibliothèque Perl, probablement

/usr/local/lib/perl, suivi de ".", pour représenter le répertoire courant. ("." ne sera pas ajouté si le mode souillé (taint mode) est activé, que ce soit par `-T` ou par `-t`.) Si vous devez modifier cette variable à l'exécution vous pouvez utiliser `use lib` pour charger les bibliothèques dépendant de la machine :

```
use lib '/mypath/libdir/';
use SomeMod;
```

Vous pouvez aussi insérer des crochets [hooks] dans le système d'inclusion de fichiers en mettant du code Perl directement dans @INC. Ces crochets peuvent être des références à des sous-programmes, des références à des tableaux ou des objets bénis. Voir require in *perlfunc* pour les détails.

@_

Dans un sous-programme, le tableau @_ contient les paramètres passés à ce sous-programme. Voir *perlsub*.

%INC

Contient une entrée pour chacun des fichiers inclus par les opérateurs `do`, `require` ou `use`. La clef est le nom du fichier (avec les noms de modules convertis en chemin), et la valeur est la localisation du fichier effectivement trouvé. La commande `require` utilise ce hachage pour déterminer si un fichier donné a déjà été inclus.

Si le dossier a été chargé par un crochet (par exemple une référence à un sous-programme, voyez require in *perlfunc* pour une description de ces crochets), ce crochet est inséré par défaut dans %INC à la place d'un nom de fichier. Notez toutefois que le crochet a pu placer l'entrée dans %INC par lui-même pour fournir quelques infos plus spécifiques.

%ENV

$ENV{expr}

Contient les variables d'environnement. Fixer une valeur dans `ENV` change l'environnement pour les processus fils.

%SIG

$SIG{expr}

Contient les gestionnaires de divers signaux. Par exemple :

```
sub handler {       # Le premier argument est le nom du signal
    my($sig) = @_;
    print "Caught a SIG$sig--shutting down\n";
    close(LOG);
    exit(0);
}

$SIG{'INT'}  = \&handler;
$SIG{'QUIT'} = \&handler;
...
$SIG{'INT'} = 'DEFAULT';    # Restaure l'action par défaut
$SIG{'QUIT'} = 'IGNORE';    # ignore SIGQUIT
```

Donner la valeur `'IGNORE'` a habituellement pour effet d'ignorer le signal, à l'exception du signal `CHLD`. Voir *perlipc* pour en savoir plus au sujet de ce cas spécial.

Voici d'autres exemples :

```
$SIG{"PIPE"} = "Plumber";   # suppose main::Plumber (Pas recommandé)
$SIG{"PIPE"} = \&Plumber;   # Bien; suppose Plumber en sous-programme
$SIG{"PIPE"} = *Plumber;    # quelque peu ésotérique
$SIG{"PIPE"} = Plumber();   # aïe, que retourne Plumber() ??
```

Assurez-vous de ne pas utiliser de mot nu comme nom de descripteur de signal, de peur que vous ne l'appeliez par inadvertance.

Si votre système reconnaît la fonction sigaction(), alors la gestion des signaux est implémentée par cette fonction. Ce qui signifie que vous avez une gestion des signaux fiable.

Depuis la version 5.8.0, par défaut, la réception d'un signal n'est plus immédiate (comprendre "non fiable") mais différée (comprendre "fiable"). Voir *perlipc* pour plus d'information.

On peut attacher un gestionnaire à certaines interruptions internes via la table de hachage %SIG. La routine indiquée par `$SIG{__WARN__}` est appelée quand un message d'avertissement est sur le point d'être affiché. Le message est passé en premier argument. La présence de l'indicateur __WARN__ supprime les avertissements normaux sur STDERR. Vous pouvez vous en servir pour stocker les avertissements dans une variable, ou pour les transformer en erreurs fatales, comme ceci :

```
local $SIG{__WARN__} = sub { die $_[0] };
eval $proggie;
```

La routine indiquée par `$SIG{__DIE__}` est appelée juste avant la gestion d'une erreur fatale, avec le message d'erreur comme premier argument. En sortie de la routine, le traitement de l'erreur reprend son cours normal, sauf si la routine provoque un arrêt du traitement via un `goto`, une sortie de boucle ou un die(). Le gestionnaire `__DIE__` est explicitement désactivé pendant l'appel, de sorte que l'interruption `__DIE__` soit possible. Même chose pour `__WARN__`.

Dû à une erreur d'implémentation, le gestionnaire `$SIG{__DIE__}` est appelé même à l'intérieur d'un eval(). N'utilisez pas ceci pour récrire une exception en suspens dans `$@`, ou bizarrement pour surcharger CORE::GLOBAL::die(). Cette étrange action à distance devrait être supprimée dans une version future de manière que `$SIG{__DIE__}` soit appelé uniquement quand votre programme est sur le point de se terminer, ce qui était l'intention à l'origine. Tout autre usage est désapprouvé.

Note : `__DIE__`/`__WARN__` ont ceci de spécial, qu'ils peuvent être appelés pour signaler une erreur (probable) pendant l'analyse du script. Dans ce cas, l'analyseur peut être dans un état instable, et toute tentative pour évaluer du code Perl provoquera certainement une faute de segmentation. Donc un appel entraînant une analyse du code devra être utilisée avec précaution, comme ceci :

```
require Carp if defined $^S;
Carp::confess("Un problème") if defined &Carp::confess;
die "Un problème, mais je ne peux charger Carp pour les détails...
     Essayer de relancer avec l'option -MCarp";
```

Dans cet exemple, la première ligne chargera Carp *sauf* si c'est l'analyseur qui a appelé l'interruption. La deuxième ligne affichera une trace de l'échec et arrêtera le programme si Carp est disponible. La troisième ligne ne sera exécutée que si Carp n'est pas disponible.

Voir die in *perlfunc*, warn in *perlfunc* et eval in *perlfunc* pour plus d'informations.

2.2 Indicateurs d'erreur

Les variables $@, $!, $^E, et $? contiennent des informations à propos des différentes conditions d'erreur pouvant survenir pendant l'exécution d'un script Perl. Les variables sont données en fonction de la "distance" qui sépare le sous-système déclenchant l'erreur et le programme Perl. Elles correspondent respectivement aux erreurs détectées par l'interpréteur Perl, la bibliothèque C, le système d'exploitation ou un programme externe.

Pour illustrer ces différences, prenons l'exemple suivant :

```
eval q{
    open my $pipe, "/cdrom/install |" or die $!;
    my @res = <$pipe>;
    close $pipe or die "bad pipe: $?, $!";
};
```

Après l'exécution du code, les 4 variables peuvent être positionnées.

$@ le sera si l'expression à (`eval`)-uer ne s'est pas compilée (cela peut se produire si les fonctions `open` ou `close` ont été importées avec de mauvais prototypes), ou si le code Perl exécuté pendant l'évaluation échoue via die(). Dans ce cas $@ contient l'erreur de compilation, ou l'argument de `die` (qui interpolera `$!` et `$?`). (Néanmoins, voir aussi *Fatal*.)

Quand le code précédent est exécuté, open(), `<PIPE>`, et `close` sont traduits en appels à la librairie C et de là vers le noyau du système d'exploitation. La variable `$!` est positionnée à la valeur `errno` de la bibliothèque C si l'un de ces appels échoue.

Sous quelques systèmes d'exploitation, `$^E` peut contenir un indicateur d'erreur plus verbeux, tel que, dans ce cas : "Le tiroir du CDROM n'est pas fermé." Sur les systèmes qui ne supportent pas de messages d'erreur étendus, `$^E` contient la même chose que `$!`.

Finalement, `$?` sera différent de `0` si le programme externe au script `/cdrom/install` échoue. Les huit bits de poids fort reflètent les conditions spécifiques de l'erreur rencontrée par le programme (la valeur exit() du programme). Les huit bits de poids faible reflètent le mode de l'échec, comme signal d'arrêt et information de vidage mémoire. Voir wait(2) pour les détails. Contrairement à `$!` et `$^E` qui changent de valeur uniquement quand une erreur est détectée, la variable `$?` est changée à chaque `wait` ou fermeture de pipe, écrasant l'ancienne valeur. C'est un peu comme `$@` qui, sur chaque eval(), est toujours fixée en cas d'échec et vidée en cas de réussite.

Pour plus de détails, voir les descriptions individuelles de $@, $!, $^E, et $?.

2.3 Note technique sur la syntaxe de noms variables

Les noms de variables en Perl peuvent avoir plusieurs formats. Habituellement, ils doivent commencer par une lettre ou un trait-de-soulignement, auquel cas ils peuvent être arbitrairement long (jusqu'à une limite interne de 251 caractères) et peuvent contenir des lettres, des chiffres, des traits-de-soulignement, ou la séquence spéciale `::` ou encore `'`. Dans ce cas, la partie placée avant le dernier `::` ou `'` est interprétée comme un *qualificateur de paquetage*; voir *perlmod*.

Les noms de variables en Perl peuvent être aussi une séquence de chiffres ou un signe de ponctuation unique ou un caractère du contrôle. Ces noms sont tous réservés par Perl pour des usages spéciaux ; par exemple, les noms 'tout en chiffres' sont utilisés pour contenir les données capturées par les références-arrières d'une expression rationnelle après une recherche de motif réussie. Perl a une syntaxe spéciale pour les noms 'caractère de contrôle seul' : il comprend `^X` (circonflexe `X`) comme signifiant le caractère contrôle-`X`. Par exemple, la notation `$^W` (dollar circonflexe `W`) est la variable scalaire dont le nom est le seul caractère contrôle-`W`. C'est mieux que de taper un contrôle-`W` directement dans votre programme.

Finalement, ce qui est nouveau dans Perl 5.6, les noms de variables peuvent être des chaînes alphanumériques qui commencent par des caractères de contrôle (ou mieux encore, un circonflexe). Ces variables doivent être écrites sous la forme `${^Foo}`; les accolades ne sont pas facultatives. `${^Foo}` dénote la variable scalaire dont le nom est un contrôle-`F` suivi par deux `o`. Ces variables sont réservées par Perl pour de futurs usages spéciaux, à l'exception de ceux qui commencent par `^_` (contrôle-soulignement ou circonflexe-soulignement). Aucun nom du type contrôle-caractère qui commence par `^_` n'acquerra de signification spéciale dans toute future version de Perl ; de tels noms peuvent donc être utilisés sans risque dans les programmes. Toutefois `$^_` lui-même, *est* réservé.

Les identificateurs Perl qui commencent par des chiffres, des caractères de contrôle ou des caractères de ponctuation ne sont pas soumis aux effets de la déclaration `package` et sont toujours forcément dans le paquetage `main` ; ils ne sont pas non plus soumis aux erreurs de `strict 'vars'`. Quelques autres noms en sont aussi exemptés :

```
ENV         STDIN
INC         STDOUT
ARGV        STDERR
ARGVOUT     _
SIG
```

En particulier, les nouvelles variables spéciales `${^_XYZ}` sont toujours associées au paquetage `main`, indépendamment de toute déclaration de paquetage dans la portée.

3 BUGS

Dû à un accident fâcheux dans l'implémentation de Perl, `use English` impose une pénalité de performance considérable sur toutes les recherches de motif dans un programme, indépendamment de l'endroit où elles se trouvent dans la portée de `use English`. Pour cette raison, l'utilisation de `use English` est fortement déconseillée dans les bibliothèques. Voir la documentation du module Devel::SawAmpersand du CPAN (http://www.cpan.org/modules/by-module/Devel/) pour plus d'information.

Avoir à penser à la variable `$^S` dans vos gestionnaires d'exceptions est tout simplement mauvais. `$SIG{__DIE__}`, tel qu'il est implémenté actuellement, favorise le risque d'erreurs graves et difficiles à localiser. Évitez-le et utilisez un `END{}` ou CORE::GLOBAL::die à la place.

4 TRADUCTION

4.1 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/.

4.2 Traducteur

Fabien Martinet <ho.fmartinet@cma-cgm.com> (version de perl 5.00502). Mise à jour : Jean-Louis Morel <jl_morel@bribes.org>, Paul Gaborit <paul.gaborit at enstimac.fr>.

4.3 Relecture

Jean-Louis Morel <jl_morel@bribes.org>.

5 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.