perldata

Table des matières

1 NAME/NOM

perldata - Types de données de Perl

2 DESCRIPTION

2.1 Noms des variables

Perl a trois types de données intégrés : les scalaires, les tableaux de scalaires et les tableaux associatifs de scalaires, appelés aussi tables de hachage ou « hachages ». Un scalaire est soit une simple chaîne de caractères (sans limite de taille si ce n'est la mémoire disponible), soit un nombre, soit une référence à quelque chose (notion expliquée dans *perlref*). Les tableaux normaux sont des listes ordonnées de scalaires indexées par des nombres, en commençant par 0. Les tables de hachages sont des collections non ordonnées de valeurs scalaires indexées par des chaînes qui sont leurs clés associées.

On fait habituellement référence aux valeurs par leur nom, ou par une référence nommée. Le premier caractère du nom vous indique à quel type de structure de données il correspond. Le reste du nom vous dit à quelle valeur particulière il fait référence. Habituellement, ce nom est un simple *identifiant*, c'est-à-dire une chaîne commençant par une lettre ou un caractère souligné, et contenant des lettres, des soulignés, et des chiffres. Dans certains cas, il peut être une chaîne d'identifiants, séparés par `::` (ou par le légèrement archaïque `'`) ; tous sauf le dernier sont interprétés comme des noms de paquetages, pour localiser l'espace de nommage dans lequel l'identifiant final doit être recherché (voir Paquetages in *perlmod* pour plus de détails). Il est possible de substituer à un simple identifiant une expression qui produit une référence à la valeur lors de l'exécution. Ceci est décrit plus en détails plus bas, et dans *perlref*.

Perl a aussi ses propres variables intégrées dont les noms ne suivent pas ces règles. Elles ont des noms étranges pour qu'elles ne rentrent pas accidentellement en collision avec l'une de vos variables normales. Les chaînes qui correspondent aux parties entre parenthèses d'une expression rationnelle sont sauvées sous des noms qui ne contiennent que des chiffres après le `$` (voir *perlop* et *perlre*). De plus, plusieurs variables spéciales qui ouvrent des fenêtres dans le fonctionnement interne de Perl ont des noms contenant des signes de ponctuation et des caractères de contrôle. Elles sont décrites dans *perlvar*.

Les valeurs scalaires sont toujours désignées par un '$', même si l'on se réfère à un scalaire qui fait partie d'un tableau ou d'un hachage. Le symbole '$' fonctionne d'un point de vue sémantique comme les articles « le » ou « la » dans le sens où ils indiquent qu'on attend une seule valeur.

```
$days               # la simple valeur scalaire "days"
$days[28]           # le 29e élément du tableau @days
$days{'Feb'}        # la veleur 'Feb' dans le hachage %days
$#days              # le dernier indice du tableau @days
```

Les tableaux complets (et les tranches de tableaux ou de hachages) sont dénotés par '@', qui fonctionne plutôt comme le mot « ces », en ce sens qu'il indique que de multiples valeurs sont attendues .

```
@days               # ($days[0], $days[1],... $days[n])
@days[3,4,5]        # identique à ($days[3],$days[4],$days[5])
@days{'a','c'}      # identique à ($days{'a'},$days{'c'})
```

Les hachages complets sont dénotés par '%' :

```
%days               # (clé1, valeur1, clé2, valeur2 ...)
```

De plus, les sous-programmes sont nommés avec un '&' initial, bien que ce soit optionnel lorsqu'il n'y a pas d'ambiguïté, tout comme « faire » est souvent redondant en français. Les entrées des tables de symboles peuvent être nommées avec un '*' initial, mais vous ne vous souciez pas vraiment de cela pour le moment (sait-on jamais... :-)).

Chaque type de variable a son propre espace de nommage, tout comme les identifiants de plusieurs types autres que les variables. Ceci signifie que vous pouvez, sans craindre de conflit, utiliser le même nom pour une variable scalaire, un tableau, ou un hachage – mais aussi pour un handle de fichier, un handle de répertoire, un nom de sous-programme, ou un label. Ceci veut dire que $foo et @foo sont deux variables différentes. Ceci veut aussi dire que `$foo[1]` fait partie de @foo, et pas de $foo. Cela peut sembler un peu étrange, mais c'est normal, puisque c'est étrange.

Puisque les références de variables commencent toujours par '$', '@', ou '%', les mots « réservés » ne sont en fait pas réservés en ce qui concerne les noms de variables (Ils SONT toutefois réservés en ce qui concerne les labels et les handles de fichiers, qui n'ont pas de caractère spécial initial. Vous ne pouvez pas avoir un handle de fichier nommé « log », par exemple. Indice : vous pourriez dire `open(LOG,'logfile')` plutôt que `open(log,'logfile')`. Utiliser des handles de fichiers en lettres majuscules améliore aussi la lisibilité et vous protège de conflits avec de futurs mots réservés. La casse *est* significative – « FOO », « Foo », et « foo » sont tous des noms différents. Les noms qui commencent par une lettre ou un caractère souligné peuvent aussi contenir des chiffres et des soulignés.

Il est possible de remplacer un tel nom alphanumérique par une expression qui retourne une référence au type approprié. Pour une description de ceci, voir *perlref*.

Les noms qui commencent par un chiffre ne peuvent contenir que des chiffres. Les noms qui ne commencent pas par une lettre, un souligné, un chiffre ou un accent circonflexe (donc un caractère de contrôle) sont limités à un caractère, tels `$%` or `$$` (La plupart de ces noms d'un seul caractère ont une signification prédéfinie pour Perl. Par exemple, `$$` est l'ID du processus courant).

2.2 Contexte

L'interprétation des opérations et des valeurs en Perl dépend parfois des exigences du contexte de l'opération ou de la valeur. Il existe deux contextes majeurs : le contexte de liste et le contexte scalaire. Certaines opérations retournent des valeurs de liste dans les contextes qui réclament une liste, et des valeurs scalaires autrement Ssi ceci est vrai pour une opération alors cela sera mentionné dans la documentation pour cette opération. En d'autres termes, Perl surcharge certaines opérations selon que la valeur de retour attendue est singulière ou plurielle. Certains mots en français fonctionnent aussi de cette façon, comme « lys » et « dos ».

Réciproquement, une opération fournit un contexte scalaire ou de liste à chacun de ses arguments. Par exemple, si vous dites :

```
int( <STDIN> )
```

L'opération int fournit un contexte scalaire pour l'opérateur <STDIN>, qui répond en lisant une ligne depuis STDIN et en la passant à l'opération int, qui trouvera alors la valeur entière de cette ligne et retournera cela. Si, au contraire, vous dites :

```
sort( <STDIN> )
```

alors l'opération sort fournit un contexte de liste pour <STDIN>, qui se mettra à lire toutes les lignes disponibles jusqu'à la fin du fichier, et passera cette liste de lignes à la routine de tri, qui triera alors ces lignes et les retournera en tant que liste à ce qui est le contexte de sort, quel qu'il soit.

L'affectation est un petit peu spéciale en ce sens qu'elle utilise son argument gauche pour déterminer le contexte de l'argument droit. L'affectation à un scalaire évalue la partie droite dans un contexte scalaire, tandis que l'affectation à un tableau ou à un hachage évalue la partie droite dans un contexte de liste. L'affectation à une liste (ou à une tranche, qui est juste une liste de toute façon) évalue aussi la partie droite dans un contexte de liste.

Lorsque vous utilisez le pragma `use warnings` ou l'option de ligne de commande **-w** de Perl, il arrive que vous voyiez des avertissements sur un usage inutile de constantes ou de fonctions dans un « contexte vide » (« void context », NDT). Le contexte vide signifie juste que la valeur a été abandonnée, comme pour une instruction ne contenant que `"fred";` ou `getpwuid(0);`. Il compte toujours pour un contexte scalaire pour les fonctions qui se soucient de savoir si elles sont ou non appelées dans un contexte scalaire.

Les sous-programmes définis par l'utilisateur peuvent se soucier d'avoir été appelés dans un contexte vide, scalaire ou de liste. La plupart des sous-programmes n'en ont toutefois pas besoin. C'est parce que les scalaires et les listes sont automatiquement interpolés en listes. Voir `wantarray()` dans *perlfunc* pour une façon dont vous pourriez discerner dynamiquement le contexte d'appel de votre fonction.

2.3 Valeurs scalaires

Toute donnée en Perl est un scalaire, un tableau de scalaires ou un hachage de scalaires. Les variables scalaires peuvent contenir des une seule valeur de trois formes différentes : un nombre, une chaîne ou une référence. En général, la conversion d'une forme à une autre est transparente. Bien qu'un scalaire ne puisse pas contenir des valeurs multiples, il peut contenir une référence à un tableau ou à un hachage qui à son tour contient des valeurs multiples.

Les scalaires ne sont pas nécessairement une chose ou une autre. Il n'y a pas d'endroit où déclarer qu'une variable scalaire doit être de type « chaîne », de type « nombre », de type « référence », ou n'importe quoi d'autre. Du fait de la conversion automatique des scalaires, les opérations qui en retournent n'ont pas besoin de se soucier (et en fait ne le peuvent pas) de savoir si leur appelant attend une chaîne, un nombre ou une référence. Perl est un langage contextuellement polymorphe dont les scalaires peuvent être des chaînes, des nombres, ou des références (ce qui inclut les objets). Tandis que les chaînes et les nombres sont considérés comme presque la même chose pour pratiquement tous les usages, les références sont des pointeurs au typage fort et impossible à forcer, avec comptage de référence intégré et invocation de destructeur.

Une valeur scalaire est interprétée comme TRUE (VRAIE, NDT) au sens booléen si ce n'est pas une chaîne vide ou le nombre 0 (ou son équivalent sous forme de chaîne, « 0 »). Le contexte booléen est juste un genre spécial de contexte scalaire, où aucune conversion vers une chaîne ou un nombre n'est jamais effectuée.

Il existe en fait deux variétés de chaînes nulles (parfois appelées des chaînes « vides »), l'une définie et l'autre non. La version définie est juste une chaîne de longueur zéro, telle que `""`. La version non définie est la valeur qui indique qu'il n'existe pas de vraie valeur pour quelque chose, comme lorsqu'il s'est produit une erreur, ou à la fin d'un fichier, ou lorsque vous vous référez à une variable ou à un élément de tableau ou de hachage non initialisé. Bien que dans les anciennes versions de Perl, un scalaire indéfini ait pu devenir défini lorsqu'il était utilisé pour la première fois dans un endroit où une valeur définie était attendue, cela ne se produit plus, sauf dans de rares cas d'autovivification tels qu'expliqués dans *perlref*. Vous pouvez utiliser l'opérateur defined() pour déterminer si une valeur scalaire est définie (cela n'a pas de sens pour les tableaux ou les hachages), et l'opérateur undef() pour produire une valeur indéfinie.

Pour trouver si une chaîne donnée est un nombre différent de zéro valide, il suffit parfois de la tester à la fois avec le 0 numérique et le « 0 » lexical (bien que ceci provoquera du bruit si les avertissements sont actifs). C'est parce que les chaînes qui ne sont pas des nombres comptent comme 0, tout comme en **awk** :

```
if ($str == 0 && $str ne "0")  {
    warn "That doesn't look like a number";
}
```

Cette méthode est peut-être meilleure parce qu'autrement vous ne traiteriez pas correctement les notations IEEE comme `NaN` ou `Infinity`. À d'autres moments, vous pourriez préférer déterminer si une donnée chaîne peut être utilisée numériquement en appelant la fonction POSIX::strtod() ou en inspectant votre chaîne avec une expression rationnelle (tel que documenté dans *perlre*).

```
warn "has nondigits"        if     /\D/;
warn "not a whole number"   unless /^\d+$/;
warn "not an integer"       unless /^[+-]?\d+$/
warn "not a decimal number" unless /^[+-]?\d+\.?\d*$/
warn "not a C float"
    unless /^([+-]?)(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?$/;
```

La longueur d'un tableau est une valeur scalaire. Vous pourriez trouver la longueur du tableau @days en évaluant `$#days`, comme en **csh**. Techniquement, ce n'est pas la longueur du tableau ; c'est l'indice de son dernier élément, ce qui n'est pas la même chose parce qu'il y a habituellement un élément numéro 0. Une affectation à `$#days` change véritablement la longueur du tableau. Le raccourcissement d'un tableau par cette méthode détruit les valeurs intermédiaires. L'agrandissement d'un tableau ayant précédemment été raccourci ne récupère pas les valeurs qui étaient stockées dans ces éléments (c'était le cas en Perl 4, mais nous avons dû supprimer cela pour nous assurer que les destructeurs sont bien appelés quand on s'y attend).

Vous pouvez aussi gagner un tout petit peu d'efficacité en pré-étendant un tableau qui va devenir gros (vous pouvez aussi étendre un tableau en affectant des données à un élément qui est au-delà de la fin du tableau). Vous pouvez tronquer totalement un tableau en y affectant la liste vide (). Les expressions suivantes sont équivalentes :

```
@whatever = ();
$#whatever = -1;
```

Si vous évaluez un tableau dans un contexte scalaire, cela renvoie la longueur du tableau (notez que ceci n'est pas vrai pour les listes, qui renvoient leur dernière valeur, comme l'opérateur virgule en C, et contrairement aux fonctions intégrées, qui renvoient ce qu'elles ont envie de renvoyer). Ce qui suit est toujours vrai :

```
scalar(@whatever) == $#whatever - $[ + 1;
```

La version 5 de Perl a changé la sémantique de `$[` : les fichiers qui ne fixent pas la valeur de `$[` n'ont plus besoin de s'inquiéter de savoir si un autre fichier a changé sa valeur (en d'autres termes, l'usage de `$[` est désapprouvé). Donc de façon générale, vous pouvez présumer que

```
scalar(@whatever) == $#whatever + 1;
```

Certains programmeurs choisissent d'utiliser une conversion explicite pour ne rien laisser au hasard :

```
$element_count = scalar(@whatever);
```

Si vous évaluez un hachage dans un contexte scalaire, vous obtenez faux si le hachage est vide. S'il contient une paire clé/valeur quelconque, il renvoie vrai ; plus précisément, la valeur retournée est une chaîne constituée du nombre de buckets utilisés et du nombre de buckets alloués, séparés par un signe de division. Ceci n'a d'intérêt que pour déterminer si les algorithmes de hachage (compilés) en Perl ont des performances médiocres sur vos données. Par exemple, vous mettez 10 000 trucs dans un hachage, mais l'évaluation de %HASH dans un contexte scalaire révèle « 1/16 », ce qui signifie qu'un seul des seize buckets a été touché, et contient probablement tous vos 10 000 éléments. Cela ne devrait pas se produire. L'évaluation d'une table de hachage liée (par 'tie') dans un contexte scalaire engendrera une erreur fatale puisque l'information d'utilisation de bucket pour les tables de hachage liées n'est pour l'instant pas disponible.

Vous pouvez préalloué de la mémoire pour une table de hachage en affectant une valeur à la fonction keys(). Ceci arrondi le nombre de buckets alloués à la puissance de deux directement supérieure :

```
keys(%users) = 1000; # pré-alloue 1024 buckets
```

2.4 Constructeurs de valeurs scalaires

Les littéraux numériques sont spécifiés dans un quelconque des formats suivants de nombres entiers ou à virgule flottante :

```
12345
12345.67
.23E-10             # un très petit nombre
3.14_15_92          # un nombre très important
4_294_967_296       # souligné pour la lisibilité
0xff                # hexa
0xfade_bebe         # un autre hexa
0377                # octal (des chiffres commençant par 0)
0b011011            # binaire
```

Vous pouvez utiliser le caractère underscore (souligné) dans un nombre littéral entre les chiffres pour améliorer la lisibilité. Vous pouvez, par exemple, regrouper les chiffres par trois (pour les droits d'accès Unix tel 0b110_100_100) ou par quatre (pour représenter des chiffres hexa dans 0b1010_0110) ou tout autre regroupement.

Les littéraux de chaîne sont habituellement délimités soit par des apostrophes, soit par des guillemets. Ils fonctionnent beaucoup comme dans un shell Unix standard : les littéraux de chaîne entre guillemets sont sujets aux substitutions de variables et au préfixage par barre oblique inverse ; les chaînes entre apostrophes ne le sont pas (sauf pour \' et \\). Les règles habituelles d'utilisation de la barre oblique inverse en C s'appliquent aussi bien pour créer des caractères comme la nouvelle ligne, la tabulation, etc., que sous des formes plus exotiques. Voir Opérateurs apostrophe et type apostrophe in *perlop* pour une liste.

Les représentations hexadécimales, octales ou binaires sous forme de chaînes (e.g. '0xff') ne sont pas automatiquement converties sous leur représentation entière. Les fonctions hex() et oct() font ces conversions pour vous. Voir *hex*() et *oct*() pour plus de détails.

Vous pouvez aussi inclure des « nouvelles lignes » directement dans vos chaînes, i.e., elles peuvent se terminer sur une ligne différente de celles où elles ont commencé. C'est bien joli, mais si vous oubliez votre apostrophe de fin (ou votre guillemet - NDT), l'erreur ne sera pas rapportée avant que Perl n'ait trouvé une autre ligne comportant une apostrophe, qui peut se trouver bien plus loin dans le script. La substitution de variable à l'intérieur des chaînes est limitée aux variables scalaires, aux tableaux et aux tranches de tableau ou de hachage (en d'autres termes, des noms commençant par $ ou @, suivi d'une expression optionnelle entre crochets comme indice). Le segment de code qui suit affiche « The price is $100. »

```
$Price = '$100';    # pas d'interpolation
print "The price is $Price.\n";     # interpolé
```

Il n'y a pas de double interpolation en Perl, donc '$100' restera tel quel.

Par défaut les nombres flottants interpolés dans une chaîne utilise le point (".") comme séparateur décimal. Si `use locale` est actif et que POSIX::setlocale() a été appelé, le caractère utilisé comme séparateur décimal dépend du locale LC_NUMERIC. Voir *perllocale* et *POSIX*.

Comme dans certains shells, vous pouvez mettre des accolades autour d'un nom pour le séparer des caractères alphanumériques (et du caractère souligné) qui le suivent. Vous devez aussi faire cela lorsque vous interpolez une variable dans une chaîne pour séparer son nom d'un deux-points ou d'une apostrophe, puisqu'ils seraient autrement traités comme un séparateur de paquetage :

```
$who = "Larry";
print PASSWD "${who}::0:0:Superuser:/:/bin/perl\n";
print "We use ${who}speak when ${who}'s here.\n";
```

Sans les accolades, Perl aurait cherché un $whospeak, un `$who::0`, et une variable `who's`. Les deux dernières auraient été les variables $0 et $s dans le paquetage `who` (probablement) inexistant.

En fait, un identifiant situé entre de telles accolades est forcé d'être une chaîne, tout comme l'est tout identificateur isolé à l'intérieur d'un indice d'un hachage. Aucun des deux n'a besoin d'apostrophes. Notre exemple précédent, `$days{'Feb'}` peut être écrit sous la forme `$days{Feb}` et les apostrophes seront présumées automatiquement. Mais tout ce qui est plus compliqué dans l'indice sera interprété comme étant une expression. Cela signifie par exemple que `$version{2.0}` sera interprété comme `$version{2}` et non comme `$version{'2.0'}`.

2.4.1 Chaîne de version

NOTE : les chaînes de version (vstrings) sont dépréciées. Elle seront supprimées dans l'une des versions qui suivront Perl 5.8.1. Le bénéfice minime des chaînes de version est largement inférieur aux mauvaises surprises et confusions potentielles qu'elles entraînent.

Un littéral de la forme `v1.20.300.4000` est analysé comme une chaîne composée de caractères correspondants aux ordinaux spécifiés. Cette formulation, connu sous le nom v-string ou chaîne de version, fournit une façon plus lisible de construire des chaînes, au lieu d'utiliser l'interpolation parfois un peu moins lisible `"\x{1}\x{14}\x{12c}\x{fa0}"`. C'est utile pour représenter des chaînes Unicode, et pour comparer des numéros de version en utilisant les opérateurs de comparaison de chaînes, `cmp`, `gt`, `lt`, etc. Si le littéral contient plusieurs points, le `v` initial peut être omis.

```
print v9786;              # affiche le SMILEY codé en UTF-8,
                          # "\x{263a}"
print v102.111.111;       # affiche "foo"
print 102.111.111;        # idem
```

De tels littéraux sont acceptés à la fois par `require` et `use` pour réaliser une vérification de numéro de version. La variable spéciale `$^V` contient aussi le numéro de version sous cette forme de l'interpréteur Perl en cours d'utilisation. Voir $^V in *perlvar*. Notez que l'utilisation des chaînes de version pour construire des adresses IPv4 n'est pas portable à moins d'utiliser les fonctions inet_aton() et inet_ntoa() du package Socket.

Notez que depuis Perl 5.8.1, une chaîne de version à un seul nombre (comme `v65`) n'en est pas une si elle est avant l'opérateur => (qui est utilisé habituellement pour séparer une clé d'une valeur dans une table de hachage). C'était considéré comme une chaîne de version depuis Perl 5.6.0 jusqu'à Perl 5.8.0 mais cela a provoqué plus de confusion et d'erreurs que d'avantages. En revanche, les chaînes de version multi-nombres comme `v65.66` et `65.66.67` le restent bien.

2.4.2 Littéraux spéciaux

Les littéraux spéciaux __FILE__, __LINE__, et __PACKAGE__ représentent le nom de fichier courant, le numéro de la ligne, et le nom du paquetage à ce point de votre programme. Ils ne peuvent être utilisés que comme des mots-clé isolés ; ils ne seront pas interpolés dans les chaînes. S'il n'existe pas de paquetage courant (à cause d'une directive `package;`), __PACKAGE__ est la valeur indéfinie.

Les deux caractères de contrôle ^D et ^Z, et les mots-clé __END__ et __DATA__ peuvent être utilisés pour indique la fin logique d'un script avant la fin effective du fichier. Tout texte les suivant est ignoré.

Le texte qui suit __DATA__ peut être lu via le handle de fichier `PACKNAME::DATA`, où `PACKNAME` est le paquetage qui était courant lorsque le mot-clé __DATA__ a été rencontré. Le handle de fichier est laissé ouvert, pointant vers le contenu après __DATA__. Il est de la responsabilité du programme d'effectuer un `close DATA` lorsqu'il a fini d'y lire. Pour la compatibilité avec d'anciens scripts écrits avant que __DATA__ ne soit introduit, __END__ se comporte comme __DATA__ dans le script principal (mais pas dans les fichiers chargés par `require` ou `do`) et laisse le contenu restant du fichier accessible via `main::DATA`.

Voir *SelfLoader* pour une plus longue description de __DATA__, et un exemple de son utilisation. Notez que vous ne pouvez pas lire depuis le handle de fichier DATA dans un bloc BEGIN : ce bloc est exécuté dès qu'il est vu (pendant la compilation), à un moment où le mot-clé __DATA__ (ou __END__) correspondant n'a pas encore été rencontré.

2.4.3 Mots simples (ou « barewords »)

Un mot qui n'a aucune autre interprétation dans la grammaire sera traité comme s'il était une chaîne entre apostrophes. Ces mots sont connus sous le nom de « barewords ». Comme pour les handles de fichier et les labels, un bareword constitué entièrement de lettres minuscules risque d'entrer en conflit avec de futurs mots réservés, et si vous utilisez le pragma `use warnings` ou l'option **-w**, Perl vous avertira pour chacun d'entre eux. Certaines personnes pourraient vouloir rendre les barewords totalement hors-la-loi. Si vous dites

```
use strict 'subs';
```

alors tout bareword qui ne serait PAS interprété comme un appel à un sous-programme produit à la place une erreur au moment de la compilation. La restriction continue jusqu'à la fin du bloc qui le contient. Un bloc interne pourrait annuler ceci en disant `no strict 'subs'`.

2.4.4 Délimiteur de jointure des tableaux

Les tableaux et les tranches sont interpolés dans les chaînes entre guillemets en joignant tous les éléments avec le délimiteur spécifié dans la variable `$"` (`$LIST_SEPARATOR` si "use English" est spécifié), un espace par défaut. Les expressions suivantes sont équivalentes :

```
$temp = join($", @ARGV);
system "echo $temp";

system "echo @ARGV";
```

À l'intérieur d'un motif de recherche (qui subit aussi la substitution entre guillemets) il y a une malheureuse ambiguïté : est-ce que `/$foo[bar]/` doit être interprété comme `/${foo}[bar]/` (où `[bar]` est une classe de caractères pour l'expression régulière) ou comme `/${foo[bar]}/` (où `[bar]` est un indice du tableau @foo) ? Si @foo n'existe pas par ailleurs, alors c'est évidemment une classe de caractères. Si @foo existe, Perl choisit de deviner la valeur de `[bar]`, et il a presque toujours raison. S'il se trompe, ou si vous êtes simplement complètement paranoïaque, vous pouvez forcer l'interprétation correcte avec des accolades comme ci-dessus.

Si vous cherchez comment utiliser les here-documents, les informations qui étaient précédemment ici sont maintenant dans Opérateurs apostrophe et type apostrophe in *perlop*.

2.5 Constructeurs de listes de valeurs

Les valeurs de liste sont notées en séparant les valeurs individuelles par des virgules (et en enfermant la liste entre parenthèses lorsque la précédence le requiert) :

```
(LIST)
```

Dans un contexte qui ne requiert pas une valeur de liste, la valeur de ce qui apparaît être un littéral de liste est simplement la valeur de l'élément final, comme avec l'opérateur virgule en C. Par exemple,

```
@foo = ('cc', '-E', $bar);
```

affecte la totalité de la valeur de liste au tableau @foo, mais

```
$foo = ('cc', '-E', $bar);
```

affecte la valeur de la variable $bar à la variable $foo. Notez que la valeur d'un véritable tableau dans un contexte scalaire est la longueur du tableau ; ce qui suit affecte la valeur 3 à $foo :

```
@foo = ('cc', '-E', $bar);
$foo = @foo;                # $foo prend la valeur 3
```

Vous pouvez avoir une virgule optionnelle avant la parenthèse fermante d'un littéral de liste, vous pouvez donc dire :

```
@foo = (
    1,
    2,
    3,
);
```

Pour utiliser un here-document afin d'affecter un tableau, une ligne par élément, vous pourriez utiliser l'approche suivante :

```
@sauces = <<End_Lines =~ m/(\S.*\S)/g;
    normal tomato
    spicy tomato
    green chile
    pesto
    white wine
End_Lines
```

Les LIST font une interpolation automatique des sous-listes. C'est-à-dire que lorsqu'une LIST est évaluée, chaque élément de la liste est évalué dans un contexte de liste, et la valeur de liste résultante est interpolée en LIST tout comme si chaque élément était un membre de LIST. Ainsi, les tableaux perdent leur identité dans une LIST - la liste

```
(@foo,@bar,&SomeSub)
```

contient tous les éléments de @foo suivis par tous les éléments de @bar, suivis par tous les éléments retournés par le sous-programme appelé SomeSub quand il est appelé dans un contexte de liste. Pour faire une référence à une liste qui *NE* soit *PAS* interpolée, voir *perlref*.

La liste vide est représentée par (). L'interpoler dans une liste n'a aucun effet. Ainsi, ((),(),()) est équivalent à (). De façon similaire, interpoler un tableau qui ne contient pas d'élément est exactement comme si aucun tableau n'est interpolé à cet endroit-là.

L'interpolcation combine l'optionalité des parenthèses ouvrantes et fermantes (sauf quand la précédence l'impose) et la possibilité de terminer une liste par une virgule optionnelle pour accepter comme syntaxiquement légal plusieurs virgules successives dans une liste. La liste `1,,3` est la concaténation des deux listes `1,` et `3` et la première possède une virgule optionnelle. `1,,3` est `(1,),(3)` et donc `1,3` (c'est la même chose pour `1,,,3` qui est `(1,),(,),3` et donc `1,3` et ainsi de suite). Ceci n'est pas un encouragement vers l'illisibilité...

Une valeur de liste peut aussi être indicée comme un tableau normal. Vous devez mettre la liste entre parenthèses pour éviter les ambiguïtés. Par exemple :

```
# Stat renvoit une valeur de liste.
$time = (stat($file))[8];

# ICI, ERREUR DE SYNTAXE.
$time = stat($file)[8];  # OOPS, OUBLI DES PARENTHESES

# Trouver un chiffre hexadécimal.
$hexdigit = ('a','b','c','d','e','f')[$digit-10];

# Un "opérateur virgule inversé".
return (pop(@foo),pop(@foo))[0];
```

Les listes ne peuvent être affectées que si chaque élément de la liste peut l'être lui aussi :

```
($a, $b, $c) = (1, 2, 3);

($map{'red'}, $map{'blue'}, $map{'green'}) = (0x00f, 0x0f0, 0xf00);
```

Une exception à ceci est que vous pouvez affecter `undef` dans une liste. C'est pratique pour balancer certaines valeurs de retour d'une fonction :

```
($dev, $ino, undef, undef, $uid, $gid) = stat($file);
```

L'affectation de liste dans un contexte scalaire renvoie le nombre d'éléments produits par l'expression du côté droit de l'affectation :

```
$x = (($foo,$bar) = (3,2,1));       # met 3 dans $x, pas 2
$x = (($foo,$bar) = f());           # met le nombre de valeurs
                                    # de retour de f() dans $x
```

Ceci est pratique lorsque vous voulez faire une affectation de liste dans un contexte booléen, parce que la plupart des fonctions de liste renvoient une liste vide quand elle se terminent, ce qui donne un 0 quand on l'affecte, 0 qui est interprété comme FALSE (FAUX - NDT).

Ça fournit aussi un moyen idiomatique d'exécuter une fonction ou une opération dans un contexte de liste puis de compter le nombre de valeurs retournées, en assignant le résultat à une liste vide puis en l'affectant dans un contexte scalaire. Par exemple, le code suivant :

```
$count = () = $string =~ /\d+/g;
```

placera dans $count le nombre de groupes de chiffres trouvés dans $string. Cela se passe comme ça parce que la recherche de motifs a lieu dans un contexte de liste (puisqu'on l'affecte à la liste vide) et retournera donc la liste de toutes les parties reconnues dans la chaîne. L'affectation de la liste dans un contexte scalaire convertira cette liste en son nombre d'éléments (ici, le nombre de reconnaissances du motif) et ce nombre sera affecté à $count. Remarquez que l'utilisation de :

```
$count = $string =~ /\d+/g;
```

ne fonctionne pas puisque la recherche de motifs est dans un contexte scalaire et retourne donc juste vrai ou faux au lieu du nombre de reconnaissances.

L'élément final d'une liste d'affectation peut être un tableau ou un hachage :

```
($a, $b, @rest) = split;
local($a, $b, %rest) = @_;
```

Vous pouvez en vérité mettre un tableau ou un hachage n'importe où dans la liste, mais le premier situé dans la liste va aspirer toutes les valeurs, et tout ce qui le suivra deviendra indéfini. Cela peut être pratique dans un local() ou un my().

Un hachage peut être initialisé en utilisant une liste de littéraux contenant des paires d'éléments qui doivent être interprétées comme des couples clé/valeur :

```
    # identique à l'affectation de map ci-dessus
    %map = ('red',0x00f,'blue',0x0f0,'green',0xf00);
```

Tandis que les littéraux de liste et les tableaux nommés sont souvent interchangeables, ce n'est pas le cas pour les hachages. Le simple fait que vous puissiez indicer une valeur de liste comme un tableau normal ne veut pas dire que vous pouvez indicer une valeur de liste comme un hachage. De la même manière, les hachages inclus comme parties d'autres listes (y compris les listes de paramètres et les listes de retour de fonctions) s'aplatissent toujours en paires clé/valeur. C'est pourquoi il est parfois bon d'utiliser des références.

Il est parfois plus lisible d'utiliser l'opérateur => dans les paires clé/valeur. L'opérateur => est principalement le synonyme d'une virgule mais visuellement plus clair. Il permet aussi à son opérande de gauche d'être interprété comme une chaîne, si c'est un bareword qui serait un identifiant légal (=> n'agit tout de même pas sur les identifiants composés, ceux qui contiennent des deux-points). Cela rend plus jolie l'initialisation des hachages :

```
    %map = (
                red   => 0x00f,
                blue  => 0x0f0,
                green => 0xf00,
);
```

ou pour initialiser les références de hachage devant être utilisées en tant qu'enregistrements :

```
    $rec = {
                witch => 'Mable the Merciless',
                cat   => 'Fluffy the Ferocious',
                date  => '10/31/1776',
    };
```

ou pour utiliser l'appel par variables pour les fonctions compliquées :

```
$field = $query->radio_group(
            name      => 'group_name',
            values    => ['eenie','meenie','minie'],
            default   => 'meenie',
            linebreak => 'true',
            labels    => \%labels
);
```

Notez que ce n'est pas parce qu'un hachage est initialisé dans un certain ordre qu'il ressortira dans cet ordre. Voir `sort()` pour des exemples sur la façon de s'arranger pour obtenir des sorties ordonnées.

2.6 Indices

L'accès aux éléments d'un tableau s'effectue en écrivant le symbole dollar (`$`) puis le nom du tableau (sans le `@` initial) puis l'indice de l'élément voulu entre crochets. Par exemple :

```
@tableau = (5, 50, 500, 5000);
print "L'élément numéro 2 est : ", $tableau[2], "\n";
```

Les indices dans les tableaux commencent à 0. Un indice négatif retrouve les éléments en partant de la fin du tableau. Dans notre exemple, `$tableau[-1]` vaut 5000 et `$tableau[-2]` vaut 500.

L'accès aux éléments d'une table de hachage est similaire sauf qu'on utilise des accolades à la place des crochets. Par exemple :

```
%scientifiques =
(
  "Newton" => "Isaac",
  "Einstein" => "Albert",
  "Darwin" => "Charles",
  "Feynman" => "Richard",
);

print "Le prénom de Darwin est ", $scientifiques{"Darwin"}, "\n";
```

2.7 Tranches

Une façon commune d'accéder à un tableau ou à un hachage est d'en prendre un élément à la fois. Vous pouvez aussi indicer une liste pour en obtenir un seul élément.

```
$whoami = $ENV{"USER"};             # un élément du hachage
$parent = $ISA[0];                  # un élément du tableau
$dir    = (getpwnam("daemon"))[7];  # idem, mais avec une liste
```

Une tranche accède à plusieurs éléments d'une liste, d'un tableau ou d'un hachage simultanément en utilisant une liste d'indices. C'est plus pratique que d'écrire les éléments individuellement sous la forme d'une liste de valeurs scalaires séparées.

```
($him, $her)   = @folks[0,-1];              # tranche de tableau
@them          = @folks[0 .. 3];            # tranche de tableau
($who, $home)  = @ENV{"USER", "HOME"};      # tranche de hachage
($uid, $dir)   = (getpwnam("daemon"))[2,7]; # tranche de liste
```

Puisque vous pouvez affecter à une liste de variables, vous pouvez aussi affecter à une tranche de tableau ou de hachage.

```
@days[3..5]    = qw/Wed Thu Fri/;
@colors{'red','blue','green'}
               = (0xff0000, 0x0000ff, 0x00ff00);
@folks[0, -1]  = @folks[-1, 0];
```

Les affectations précédentes sont exactement équivalents à

```
($days[3], $days[4], $days[5]) = qw/Wed Thu Fri/;
($colors{'red'}, $colors{'blue'}, $colors{'green'})
               = (0xff0000, 0x0000ff, 0x00ff00);
($folks[0], $folks[-1]) = ($folks[-1], $folks[0]);
```

Puisque changer une tranche change le tableau ou le hachage original dont la tranche est issue, une structure `foreach` altèrera certaines – ou même toutes les – valeurs du tableau ou du hachage.

```
foreach (@array[ 4 .. 10 ]) { s/peter/paul/ }

foreach (@hash{qw[key1 key2]}) {
    s/^\s+//;           # supprime les espaces au début des éléments
    s/\s+$//;           # supprime les espaces à la fin des éléments
    s/(\w+)/\u\L$1/g;   # met une majuscule aux mots
}
```

Une tranche d'une liste vide est encore une liste vide. Ainsi :

```
@a = ()[1,0];           # @a n'a pas d'éléments
@b = (@a)[0,1];         # @b n'a pas d'éléments
@c = (0,1)[2,3];        # @c n'a pas d'éléments
```

Mais :

```
@a = (1)[1,0];          # @a a deux éléments
@b = (1,undef)[1,0,2];  # @b a trois éléments
```

Ceci rend aisée l'écriture de boucles qui se terminent lorsqu'une liste nulle est renvoyée :

```
while ( ($home, $user) = (getpwent)[7,0]) {
    printf "%-8s %s\n", $user, $home;
}
```

Comme noté précédemment dans ce document, le sens scalaire de l'affectation de liste est le nombre d'éléments de la partie droite de l'affectation. La liste nulle ne contient pas d'éléments, donc lorsque le fichier de mots de passe est vidé, le résultat est 0 et non pas 2.

Si vous êtes troublé par le pourquoi de l'usage d'un '@' ici sur une tranche de hachage au lieu d'un '%', pensez-y ainsi. Le type de parenthésage (avec des crochets ou des accolades) décide si c'est un tableau ou un hachage qui est examiné. D'un autre côté, le symbole en préfixe ('$' ou '@') du tableau ou du hachage indique si vous récupérez une valeur simple (un scalaire) ou une valeur multiple (une liste).

2.8 Typeglobs et handles de Fichiers

Perl utilise un type interne appelé un *typeglob* pour contenir une entrée complète de table de symbole. Le préfixe de type d'un typeglob est une *, parce qu'il représente tous les types. Ceci fut la manière favorite de passer par référence à une fonction des tableaux et des hachages, mais maintenant que nous avons de vraies références, c'est rarement nécessaire.
Le principal usage des typeglobs dans le Perl moderne est de créer des alias de table de symbole. Cette affectation :

```
*this = *that;
```

fait de $this un alias de $that, @this un alias de @that, %this un alias de %that, &this un alias de &that, etc. Il est bien plus sûr d'utiliser une référence. Ceci :

```
local *Here::blue = \$There::green;
```

fait temporairement de $Here::blue un alias de $There::green, mais ne fait pas de @Hzere::blue un alias de @There::green, ou de %Here::blue un alias de %There::green, etc. Voir Tables de Symboles in *perlmod* pour plus d'exemples de ceci. Aussi étrange que cela puisse paraître, c'est la base de tout le système d'import/export de module.
Un autre usage des typeglobs est le passage de handles de fichiers à une fonction, ou la création de nouveaux handles de fichiers. Si vous avez besoin d'utiliser un typeglob pour sauvegarder un handle de fichier, faites-le de cette façon :

```
$fh = *STDOUT;
```

ou peut-être comme une vraie référence, comme ceci :

```
$fh = \*STDOUT;
```

Voir *perlsub* pour des exemples d'usages de ceci comme handles de fichiers indirects dans des fonctions.
Les typeglobs sont aussi une façon de créer un handle de fichier local en utilisant l'opérateur local(). Ceux-ci ne durent que jusqu'à ce que leur bloc soit terminé, mais peuvent être passés en retour. Par exemple :

```
sub newopen {
    my $path = shift;
    local  *FH;  # not my!
    open   (FH, $path) || return undef;
    return *FH;
}
$fh = newopen('/etc/passwd');
```

Maintenant que nous avons la notation `*foo{THING}`, les typeglobs ne sont plus autant utilisés pour les manipulations de handles de fichiers, même s'ils sont toujours nécessaires pour passer des fichiers tout neufs et des handles de répertoire dans les fonctions. C'est parce que `*HANDLE{IO}` ne fonctionne que si HANDLE a déjà été utilisé en tant que handle. En d'autres termes, `*FH` doit être utilisé pour créer de nouvelles entrées de table de symboles ; `*foo{THING}` ne le peut pas. En cas de doute, utilisez `*FH`.
Toutes les fonctions qui sont capables de créer des handles de fichiers (open(), opendir(), pipe(), socketpair(), sysopen(), socket(), et accept()) créent automatiquement un handle de fichier anonyme si le handle qui leur est passé est une variable scalaire non initialisée. Ceci permet aux constructions telles que `open(my $fh, ...)` et `open(local $fh,...)` d'être utilisées pour créer des handles de fichiers qui seront convenablement et automatiquement fermés lorsque la portée se termine, pourvu qu'il n'existe aucune autre référence vers eux. Ceci élimine largement le besoin pour les typeglobs lors de l'ouverture des handles de fichiers qui doivent être passés à droite et à gauche, comme dans l'exemple suivant :

```
sub myopen {
    open my $fh, "@_"
         or die "Can't open '@_': $!";
    return $fh;
}

{
    my $f = myopen("</etc/motd");
    print <$f>;
    # $f implicitly closed here
}
```

Notez que si un scalaire déjà initialisé est utilisé, cela a un effet différent : `my $fh='zzz'; open($fh, ...)` est équivalent à `open( *{'zzz'}, ...)`. `use strict 'refs'` interdit cela.
Une autre façon de créer des handles de fichiers anonymes est d'utiliser le module Symbol ou le module IO::Handle et ceux de son acabit. Ces modules ont l'avantage de ne pas cacher les différents types du même nom pendant le local(). Voir la fin de la documentation de la fonction `open()` pour un exemple.

3 VOIR AUSSI

Voir *perlvar* pour une description des variables intégrées de Perl et une discussion des noms de variable légaux. Voir *perlref*, *perlsub*, et Tables de Symboles in *perlmod* pour plus de détails sur les typeglobs et la syntaxe `*foo{THING}`.

4 TRADUCTION

4.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez http://perl.enstimac.fr/.

4.2 Traducteur

Traduction initiale : Roland Trique <*roland.trique@free.fr*>. Mise à jour : Paul Gaborit *paul.gaborit at enstimac.fr*.

4.3 Relecture

Jim Fox <*fox@sugar-land.dowell.slb.com*>, Etienne Gauthier <*egauthie@capgemini.fr*>, Gérard Delafond

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.