perlfaq7

Table des matières

1 NAME/NOM

perlfaq7 - Questions générales sur le langage Perl

2 DESCRIPTION

Cette section traite des questions générales sur le langage Perl qui ne trouvent leur place dans aucune autre section.

2.1 Puis-je avoir une BNF/yacc/RE pour le langage Perl ?

Il n'y a pas de BNF, mais vous pouvez vous frayer un chemin dans la grammaire de yacc dans perly.y au sein de la distribution source si vous êtes particulièrement brave. La grammaire repose sur un code de mots-clés complexe, préparez-vous donc à vous aventurer aussi dans toke.c.

Selon les termes de Chaim Frenkel : "La grammaire de Perl ne peut pas être réduite à une BNF. Le travail d'analyse de perl est distribué entre yacc, l'analyser lexical, de la fumée et des miroirs".

2.2 Quels sont tous ces $@%&* de signes de ponctuation, et comment savoir quand les utiliser ?

Ce sont des spécificateurs, comme détaillé dans *perldata* :

```
$ pour les valeurs scalaires (nombres, chaînes ou références)
@ pour les tableaux
% pour les hachages (tableaux associatifs)
& pour les sous-programmes (alias les fonctions, les procédures,
  les méthodes)
* pour tous les types de la table des noms courante. Dans la
  version 4, vous les utilisiez comme des pointeurs, mais dans les
  versions modernes de perl, vous pouvez utiliser simplement les
  références.
```

Les trois autres symboles que vous risquez fortement de rencontrer sans qu'ils soient véritablement des spécificateurs de type sont :

```
<> utilisés pour récupérer un enregistrement depuis un
   filehandle.
\  prend une référence à quelque chose.
```

Notez que <FICHIER> n'est *ni* le spécificateur de type pour les fichiers *ni* le nom du handle. C'est l'opérateur <> appliqué au handle FICHIER. Il lit une ligne (en fait un enregistrement – voir $/ in *perlvar*) depuis le handle FICHIER dans un contexte scalaire, ou *toutes* les lignes dans un contexte de liste. Lorsqu'on ouvre, ferme ou réalise n'importe quelle autre opération à part <> sur des fichiers, ou même si l'on parle du handle, n'utilisez *pas* les crochets. Ces expressions sont correctes : `eof(FH)`, `seek(FH, 0, 2)` et "copie de STDIN vers FICHIER".

2.3 Dois-je toujours/jamais mettre mes chaînes entre guillemets ou utiliser les points-virgules et les virgules ?

Normalement, un bareword n'a pas besoin d'être mis entre guillemets, mais dans la plupart des cas, il le devrait probablement (et doit l'être sous `use strict`). Mais une clé de hachage constituée d'un simple mot (qui ne soit pas le nom d'un sous-programme) et l'opérande de gauche de l'opérateur =>, comptent tous les deux comme s'ils étaient entre guillemets :

```
Ceci                  équivaut à ceci
------------          ---------------
$foo{line}            $foo{'line'}
bar => stuff          'bar' => stuff
```

Le point-virgule final dans un bloc est optionnel, tout comme la virgule finale dans une liste. Un bon style (voir *perlstyle*) préconise de les mettre sauf dans les expressions d'une seule ligne :

```
if ($whoops) { exit 1 }
@nums = (1, 2, 3);

if ($whoops) {
    exit 1;
}
@lines = (
    "There Beren came from mountains cold",
    "And lost he wandered under leaves",
);
```

2.4 Comment ignorer certaines valeurs de retour ?

Une façon est de traiter les valeurs de retour comme une liste et de les indexer :

```
    $dir = (getpwnam($user))[7];
```

Une autre façon est d'utiliser undef comme un élément de la partie gauche :

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

Vous pouvez aussi utiliser une tranche de liste pour sélectionner uniquement les éléments nécessaires :

```
($dev, $ino, $uid, $gid) = ( stat($file) )[0,1,4,5];
```

2.5 Comment bloquer temporairement les avertissements ?

Si vous utilisez Perl 5.6.0 ou supérieur, le pragma `use warnings` vous permet un contrôle fin des avertissements qui sont générés. Voir *perllexwarn* pour plus de détails.

```
{
    no warnings;          # désactive temporairement les avertissements
    $a = $b + $c;         # je sais qu'elles peuvent être indéfinies
}
```

De plus, vous pouvez activer ou désactiver les avertissements par catégories. Vous désactivez les catégories que vous sohaitez ignorer tout en conservant les autres catégories d'avertissements. Voir *perllexwarn* pour tous les détails dont les noms des catégories et leur hiérarchie.

```
{
  no warnings 'uninitialized';
  $a = $b + $c;
}
```

Si vous avez une version plus ancienne de Perl, la variable `$^W` (documentée dans *perlvar*) contrôle les avertissements lors de l'exécution d'un bloc :

```
{
    local $^W = 0;        # supprimer temporairement les
                            avertissements
    $a = $b + $c;         # je sais que ces variables sont
                            peut-être indéfinies
}
```

Notez que comme toutes les variables de ponctuation, vous ne pouvez pas utiliser my() sur `$^W`, uniquement local().

2.6 Qu'est-ce qu'une extension ?

Une extensions est une façon d'appeler depuis Perl du code C compilé. Lire *perlxstut* est une bonne façon d'en apprendre plus sur les extensions.

2.7 Pourquoi les opérateurs de Perl ont-ils une précédence différente de celle des opérateurs en C ?

En fait, ce n'est pas le cas. Tous les opérateurs C que Perl copie ont la même précédence en Perl qu'en C. Le problème est avec les opérateurs que C ne possède pas, en particulier les fonctions qui donnent un contexte de liste à tout ce qui se trouve à leur droite, par exemple print, chmod, exec et ainsi de suite. De telles fonctions sont appelées des "opérateurs de liste" et apparaissent en tant que tels dans la table de précédence de *perlop*.

Une erreur commune est d'écrire :

```
unlink $file || die "snafu";
```

Qui est interprété ainsi :

```
unlink ($file || die "snafu");
```

Pour éviter ce problème, placez soit des parenthèses supplémentaires ou utilisez l'opérateur de précédence super basse `or` :

```
(unlink $file) || die "snafu";
unlink $file or die "snafu";
```

Les opérateurs "anglais" (`and`, `or`, `xor`, et `not`) ont une précédence délibérément plus basse que celle des opérateurs de liste juste pour des situations telles que ci-dessus.

Un autre opérateur ayant une précédence surprenante est l'exponentiation. Il se lie encore plus étroitement que même le moins unaire, faisant du produit `-2**2` un quatre négatif et non pas positif. Il s'associe aussi à droite, ce qui signifie que `2**3**2` vaut deux à la puissance neuvième, et non pas huit au carré.

Bien qu'il ait la même précédence qu'en C, l'opérateur `?:` de Perl produit une lvalue. Ce qui suit affecte $x soit à $a soit à $b, selon la valeur de vérité de $maybe :

```
($maybe ? $a : $b) = $x;
```

2.8 Comment déclarer/créer une structure ?

En général, vous ne "déclarez" pas une structure. Utilisez juste une référence de hachage (probablement anonyme). Voir *perlref* et *perldsc* pour plus de détails. Voici un exemple :

```
$person = {};                   # nouveau hachage anonyme
$person->{AGE}  = 24;           # fixe le champ AGE à 24
$person->{NAME} = "Nat";        # fixe le champ NAME à "Nat"
```

Si vous recherchez quelque chose d'un peu plus rigoureux, essayez *perltoot*.

2.9 Comment créer un module ?

(contribution de brian d foy)

perlmod, *perlmodlib* et *perlmodstyle* expliquent les modules dans tous leurs détails. *perlnewmod* fournit un bref aperçu de la manière de procéder avec quelques suggestions de style.

Si vous avez besoin d'inclure du code C ou une interface vers une bibliothèque C dans votre module, vous aurez besoin de h2xs. h2xs créera pour vous la structure de distribution de votre module et les fichiers d'interface initiaux dont vous aurez besoin. *perlxs* et *perlxstut* donnent tous les détails.

Si vous n'avez pas besoin de code C, d'autres outils comme ExtUtils::ModuleMaker et Module::Starter peuvent vous aider à créer le squelette de distribution de votre module.

Vous pouvez aussi lire "Writing Perl Modules! for CPAN" (http://apress.com/book/bookDisplay.html?bID=14) par Sam Tregar qui est le meilleur guide pour créer une distribution de module.

2.10 Comment créer une classe ?

Voir *perltoot* pour une introduction aux classes et aux objets, ainsi que *perlobj* et *perlbot*.

2.11 Comment déterminer si une variable est souillée ?

Vous pouvez utiliser la fonction tainted() du module Scalar::Util disponible sur CPAN (et inclus dans la distribution Perl depuis la version 5.8.0). Voir aussi Blanchiement et détection des données souillées in *perlsec*.

2.12 Qu'est-ce qu'une fermeture ?

Les fermetures sont documentées dans *perlref*.

La *Fermeture* est un terme d'informatique ayant un sens précis mais difficile à expliquer. Les fermetures sont implémentées en Perl par des sous-programmes anonymes ayant des références à des variables lexicales qui persistent hors de leur propre portées. Ces lexicaux se réfèrent de façon magique aux variables qui se trouvaient dans le coin lorsque le sous-programme a été défini (deep binding).

Les fermetures ont un sens dans tous les langages de programmation où la valeur de retour d'une fonction peut être une fonction, comme c'est le cas en Perl. Notez que certains langages fournissent des fonctions anonymes sans être capables de fournir des fermetures proprement dites ; le langage Python, par exemple. Pour plus d'informations sur les fermetures, regardez dans un livre sur la programmation fonctionnelle. Scheme est un langage qui non seulement supporte mais aussi encourage les fermetures.

Voici une fonction classique de génération de fonctions :

```
sub add_function_generator {
  return sub { shift() + shift() };
}

$add_sub = add_function_generator();
$sum = $add_sub->(4,5);                 # $sum vaut maintenant 9.
```

La fermeture fonctionne comme un *modèle de fonction* ayant des possibilités de personnalisation qui seront définies plus tard. Le sous-programme anonyme retourné par add_function_generator() n'est pas techniquement une fermeture car il ne se réfère à aucun lexical hors de sa propre portée.

Comparez cela à la fonction suivante, make_adder(), dans laquelle la fonction anonyme retournée contient une référence à une variable lexicale se trouvant hors de la portée de la fonction elle-mêeme. Une telle référence nécessite que Perl renvoie une fermeture proprement dite, verrouillant ainsi pour toujours la valeur que le lexical avait lorsque la fonction fut créé.

```
sub make_adder {
    my $addpiece = shift;
    return sub { shift() + $addpiece };
}

$f1 = make_adder(20);
$f2 = make_adder(555);
```

Maintenant `&$f1($n)` vaut toujours 20 plus la valeur de $n que vous lui passez, tandis que `&$f2($n)` vaut toujours 555 plus ce qui se trouve dans $n. La variable $addpiece dans la fermeture persiste.

Les fermetures sont souvent utilisées pour des motifs moins ésotériques. Par exemple, lorsque vous désirez passer un morceau de code à une fonction :

```
my $line;
timeout( 30, sub { $line = <STDIN> } );
```

Si le code à exécuter a été passé en tant que chaîne, `'$line = <STDIN>'`, la fonction hypothétique timeout() n'aurait aucun moyen d'accéder à la variable lexicale $line dans la portée de son appelant.

2.13 Qu'est-ce que le suicide de variable et comment le prévenir ?

Ce problème a été corrigé dans perl 5.004_05. Donc la prévention passe par la mise à jour de perl. ;)

Le suicide de variable se produit lorsque vous perdez la valeur d'une variable (temporairement ou de façon permanente). Il est causé par les interactions entre la portée définie par my() et local() et soit des fermetures soit des variables d'itération foreach() aliasées et des arguments de sous-programme (Note au relecteur : un peu bordélique, cette phrase). Il a été facile de perdre par inadvertance la valeur d'une variable de cette façon, mais désormais c'est bien plus difficile. Considérez ce code :

```
my $f = 'foo';
sub T {
  while ($i++ < 3) { my $f = $f; $f .= $i; print $f, "\n" }
}
T;
print "Finally $f\n";
```

Si vous constatez un suicide de variable, c'est que `my $f` dans le sous-programme ne récupère pas une nouvelle copie de `$f` dont la valeur est `foo`. La sortie suivante montre qu'à l'intérieur du sous-programme la valeur de `$f` est perdue alors qu'elle ne devrait pas :

```
foobar
foobarbar
foobarbarbar
Finally foo
```

Le $f qui se voit ajouter "bar" par trois fois devrait être un nouveau `$f`. `my $f` devrait créer une nouvelle variable locale chaque fois que la boucle est parcourue. La sortie normale est :

```
foobar
foobar
foobar
Finally foo
```

2.14 Comment passer/renvoyer {une fonction, un handle de fichier, un tableau, un hachage, une méthode, une expression rationnelle} ?

À l'exception des expressions rationnelles, vous devez passer des références à ces objets. Voir Passage par Référence in *perlsub* pour cette question particulière, et *perlref* pour plus d'informations sur les références.

Voir "Passer des expressions rationnelles", plus bas, pour le passage des expressions rationnelles.

Passer des variables et des fonctions

Les variables normales et les fonctions sont faciles à passer : passez juste une référence à une variable ou une fonction, existante ou anonyme :

```
func( \$some_scalar );

func( \@some_array  );
func( [ 1 .. 10 ]   );

func( \%some_hash   );
func( { this => 10, that => 20 }   );

func( \&some_func   );
func( sub { $_[0] ** $_[1] }   );
```

Passer des handles de fichiers

Depuis Perl 5.6, vous pouvez représenter les handles de fichiers par de simples variables scalaires qui se traitent donc comme tous les autres scalaires.

```
open my $fh, $filename or die "Cannot open $filename! $!";
func( $fh );
```

```
sub func {
  my $passed_fh = shift;
  my $line = <$passed_fh>;
}
```

Avant Perl 5.6, vous devez utiliser les notations `*FH` ou `\*FH`. Ce sont des "typeglobs" - voir Typeglobs et handles de fichiers in *perldata* et en particulier Passage par référence in *perlsub* pour plus d'information.

Passer des expressions rationnelles

Pour passer des expressions rationnelles, vous devrez utiliser une version de Perl suffisamment récente pour supporter la construction `qr//`, passer des chaînes et utiliser un eval capturant les exceptions, soit encore vous montrer très très intelligent.

Voici un exemple montrant comment passer une chaîne devant être comparée comme expression rationnelle :

```
sub compare($$) {
    my ($val1, $regex) = @_;
    my $retval = $val1 =~ /$regex/;
    return $retval;
}
$match = compare("old McDonald", qr/d.*D/i);
```

Notez comment `qr//` autorise les drapeaux terminaux. Ce motif a été compilé à la compilation, même s'il n'a été exécuté que plus tard. La sympathique notation `qr//` n'a pas été introduite avant la version 5.005. Auparavant, vous deviez approcher ce problème bien moins intuitivement. La revoici par exemple si vous n'avez pas `qr//` :

```
 sub compare($$) {
    my ($val1, $regex) = @_;
    my $retval = eval { $val1 =~ /$regex/ };
    die if $@;
    return $retval;
 }
$match = compare("old McDonald", q/($?i)d.*D/);
```

Assurez-vous de ne jamais dire quelque chose comme ceci :

```
return eval "\$val =~ /$regex/";   # FAUX
```

ou quelqu'un pourrait glisser des séquences d'échappement du shell dans l'expressions rationnelle à cause de la double interpolation de l'eval et de la chaîne doublement mise entre guillemets. Par exemple :

```
$pattern_of_evil = 'danger ${ system("rm -rf * &") } danger';
eval "\$string =~ /$pattern_of_evil/";
```

Ceux qui préfèrent être très très intelligents peuvent voir le livre O'Reilly *Mastering Regular Expressions*, par Jeffrey Friedl. La fonction Build_MatchMany_Function() à la page 273 est particulièrement intéressante. Une citation complète de ce livre est donnée dans *perlfaq2*.

Passer des Méthodes

Pour passer une méthode objet à un sous-programme, vous pouvez faire ceci :

```
call_a_lot(10, $some_obj, "methname")
sub call_a_lot {
    my ($count, $widget, $trick) = @_;
    for (my $i = 0; $i < $count; $i++) {
        $widget->$trick();
    }
}
```

Ou bien vous pouvez utiliser une fermeture pour encapsuler l'objet, son appel de méthode et ses arguments :

```
my $whatnot =  sub { $some_obj->obfuscate(@args) };
func($whatnot);
sub func {
    my $code = shift;
    &$code();
}
```

Vous pourriez aussi étudier la méthode can() de la classe UNIVERSAL (qui fait partie de la distribution standard de perl).

2.15 Comment créer une variable statique ?

(contribution de brian d foy)

Perl n'a pas de variable "statique" qui ne serait accessible que dans la fonction où elle serait déclarée. Mais vous pouvez obtenir le même effet en utilisant des variables lexicales.

Vous pouvez simuler une variable statique en utilisant une variable lexicale qui se met hors de portée. Dans l'exemple ci-dessous, nous définissons un sous-programme `compteur` qui utiliser la variable lexicale `$compte`. Puisque tout cela se fait dans un bloc BEGIN, non seulement `$compte` est défini lors de la compilation mais aussi elle se met hors de portée à la fin du bloc BEGIN. Le bloc BEGIN garantit que le sous-programme et la valeur qu'il utilise sont définis lors de la compilation et donc le sous-programme est utilisable comme n'importe quel autre sous-programme, et vous pouvez placer ce bout de code aux mêmes endroits qu'un sous-programme classique (en fin de programme par exemple). Le sous-programme `compteur` conserve une référence à la donnée et c'est le seul moyen que vous avez d'accéder à cette valeur (et à chaque fois que vous le faites, vous incrémentez la valeur). La donnée définie par `$compte` est donc une donnée privée de `compteur`.

```
BEGIN {
  my $compte = 1;
  sub compteur { $compte++ }
}

my $debut = compteur();

.... # du code qui appelle compteur();

my $fin = compteur();
```

Dans l'exemple précédent, nous avons créer une variable privée visible d'une seule fonction. Nous pourrions définir plusieurs fonctions tant que la variable reste visible et toutes ces fonctions partageraient la même variable "privée". Ce n'est plus réellement "statique" puisque plusieurs fonctions accèdent à la même variable. Dans l'exemple suivant, `incremente_compteur` et `donne_compteur` partage la variable. La première fonction ajoute 1 à la valeur alors que la seconde ne fait que retourner la valeur. Toutes deux peuvent accéder à la valeur et il n'existe pas d'autres moyen de le faire.

```
BEGIN {
  my $compte = 1;
  sub incremente_compteur { $compte++ }
  sub donne_compteur      { $compte }
}
```

Pour déclarer une variable privée d'un fichier, vous pouvez encore utiliser une variable lexicale. La portée d'une variable lexicale se limite à son contenant (un bloc ou un fichier) et donc une variable lexicale déclarée dans un fichier est invisible depuis un autre fichier.

Voir Variables privées persistantes in *perlsub* pour plus de détails. Tout ce qui est présenté dans *perlref* concernant les fermetures peut aussi vous aider même si nous n'utilisons pas de sous-programmes anonymes dans cette réponse.

2.16 Quelle est la différence entre la portée dynamique et lexicale (statique) ? Entre local() et my() ?

`local($x)` sauvegarde l'ancienne valeur de la variable globale `$x` et lui affecte une nouvelle valeur pour la durée du sous-programme, *valeur qui est visible dans les autres fonctions appelées depuis ce sous-programme*. C'est fait lors de l'exécution et donc cela s'appelle une portée dynamique. local() affecte toujours les variables globales, aussi appelées variables de paquetage ou variables dynamiques.

`my($x)` crée une nouvelle variable qui n'est visible que dans le sous-programme. C'est fait lors de la compilation et donc cela s'appelle une portée lexicale ou statique. my() affecte toujours les variables privées, aussi appelées variables lexicales ou (de façon impropre) variables (de portée) statique.

Par exemple :

```
sub visible {
    print "var has value $var\n";
}
```

```
sub dynamic {
    local $var = 'local';   # nouvelle valeur temporaire pour la
    visible();              # variable toujours globale appelée $var
}

sub lexical {
    my $var = 'private';    # nouvelle variable privée, $var
    visible();              # (invisible hors de la portée de la routine)
}

$var = 'global';

visible();                  # affiche global
dynamic();                  # affiche local
lexical();                  # affiche global
```

Notez que, comme à aucun moment la valeur "private" n'est affichée. C'est parce que $var n'a cette valeur qu'à l'intérieur du bloc de la fonction lexical(), et est cachée des sous-programmes appelés.

En résumé, local() ne crée pas comme vous pourriez le penser des variables locales, privées. Il donne seulement à une variable globale une valeur temporaire. my() est ce que vous recherchez si vous voulez des variables privées.

Voir Variables privées via my() in *perlsub* et Valeurs temporaires via local() in *perlsub* pour plus de détails.

2.17 Comment puis-je accéder à une variable dynamique lorsqu'un lexical de même nom est à portée ?

Si vous connaissez le nom du paquetage, vous pouvez tout simplement le mentionner explicitement comme dans $Un_paquetage::var. Notez que la notation $::var ne signifie **pas** la variable dynamique $var du paquetage courant mais celle du paquetage "main" exactement comme si vous aviez écrit $main::var.

```
use vars '$var';
local $var = "global";
my    $var = "lexical";

print "lexical is $var\n";
print "global  is $main::var\n";
```

Vous pouvez aussi utiliser la directive our() du compilateur pour amener un vars dynamique à portée.

```
require 5.006; # our() n'existait avant 5.6
use vars '$var';

local $var = "global";
my $var    = "lexical";

print "lexical is $var\n";

{
  our $var;
  print "global  is $var\n";
}
```

2.18 Quelle est la différence entre les liaisons profondes et superficielles ?

Dans la liaison profonde, les variables lexicales mentionnées dans les sous-programmes anonymes sont les mêmes que celles qui étaient dans la portée lorsque le sous-programme fut créé. Dans la liaison superficielle, ce sont les variables, quelles qu'elles soient, ayant le même nom et s'étant trouvé dans la portée lorsque le sous-programme est appelé. Perl utilise toujours la liaison profonde des variables lexicales (i.e., celles créées avec my()). Toutefois, les variables dynamiques (aussi appelées globales, locales ou variables de paquetage) sont effectivement liées superficiellement. Considérez ceci comme simplement une raison de plus de ne pas les utiliser. Voir la réponse à Qu'est-ce qu'une fermeture ? (§2.12).

2.19 Pourquoi "my($foo) = <FICHIER>;" ne marche pas ?

`my()` et `local()` donnent un contexte de liste à la partie droite de =. L'opération de lecture <FH>, comme tant d'autres fonctions et opérateurs de Perl, peut déterminer dans quel contexte elle a été appelée pour se comporter de façon appropriée. En général, la fonction scalar() peut aider. Cette fonction ne fait rien aux données elles-mêmes (contrairement au mythe populaire) mais dit seulement à ses arguments de se comporter à sa propre manière scalaire. Si cette fonction n'a pas de comportement scalaire défini, ceci ne vous aidera visiblement pas (tout comme avec sort()).

Pour forcer un contexte scalaire dans ce cas particulier, toutefois, vous devez tout simplement omettre les parenthèses :

```
local($foo) = <FILE>;           # FAUX
local($foo) = scalar(<FILE>);   # ok
local $foo  = <FILE>;           # bien
```

Vous devriez probablement utiliser des variables lexicales de toute façon, même si le résultat est le même ici :

```
my($foo) = <FILE>;  # FAUX
my $foo  = <FILE>;  # bien
```

2.20 Comment redéfinir une fonction, un opérateur ou une méthode prédéfini ?

Pourquoi donc voulez-vous faire cela ?

Si vous voulez surcharger une fonction prédéfinie, telle qu'open(), alors vous devrez importer la nouvelle définition depuis un module différent. Voir Surcharge des Fonctions Prédéfinies in *perlsub*. Il y a aussi un exemple dans Class::Template in *perltoot*.

Si vous voulez surcharger un opérateur de Perl, tel que + ou **, alors vous voudrez utiliser le pragma `use overload`, documentée dans *overload*.

Si vous parlez d'obscurcir les appels de méthode dans les classes parent, voyez Polymorphisme in *perltoot*.

2.21 Quelle est la différence entre l'appel d'une fonction par &foo et par foo() ?

Quand vous appelez une fonction par `&foo`, vous permettez à cette fonction d'accéder à vos valeurs @_ courantes, et vous court-circuitez les prototypes. Cela signifie que la fonction ne récupère pas un @_ vide, mais le vôtre ! Même si l'on ne peut pas exactement parler d'un bug (c'est documenté de cette façon dans *perlsub*), il serait difficile de considérer cela comme une caractéristique dans la plupart des cas.

Quand vous appelez votre fonction par `&foo()`, alors vous obtenez *effectivement* une nouvelle @_, mais le prototypage est toujours court-circuité.

Normalement, vous préférerez appeler une fonction en utilisant `foo()`. Vous ne pouvez omettre les parenthèses que si la fonction est déjà connue par le compilateur parce qu'il a déjà vu sa définition (`use` mais pas `require`), ou via une référence en avant ou une déclaration `use subs`. Même dans ce cas, vous obtenez une @_ propre sans aucune des vieilles valeurs suintant là où elles ne devraient pas.

2.22 Comment créer une instruction switch ou case ?

C'est expliqué plus en profondeur dans *perlsyn*. En bref, il n'existe pas d'instruction case officielle, à cause de la variété des tests possibles en Perl (comparaison numérique, comparaison de chaînes, comparaison de glob, appariement d'expression rationnelle, comparaisons surchargées...). Larry ne pouvait pas se décider sur la meilleure façon de faire cela, il a donc laissé tomber et c'est resté sur la liste des voeux pieux depuis perl1.

Depuis Perl 5.8, pour avoir les instruction switch et case, vous pouvez utiliser le module Switch en disant :

```
use Switch;
```

Ce n'est pas aussi rapide que cela pourrait l'être puisque ce n'est pas vraiment intégré au langage (c'est fait par du filtrage des sources) mais ça existe et c'est pratique.

Si vous préférez utiliser du Perl pur, la réponse générale est d'écrire une construction comme celle-ci :

```
for ($variable_to_test) {
    if    (/pat1/)  { }     # faire quelque chose
    elsif (/pat2/)  { }     # faire quelque chose d'autre
    elsif (/pat3/)  { }     # faire quelque chose d'autre
    else            { }     # défaut
}
```

Voici un exemple simple de switch basé sur un appariement de motif, cette fois mis en page de façon à le faire ressembler un peu plus à une instruction switch. Nous allons créer un choix multiple basé sur le type de référence stockée dans $whatchamacallit :

```
SWITCH: for (ref $whatchamacallit) {

    /^$/            && die "not a reference";

    /SCALAR/        && do {
                            print_scalar($$ref);
                            last SWITCH;
                    };

    /ARRAY/         && do {
                            print_array(@$ref);
                            last SWITCH;
                    };

    /HASH/          && do {
                            print_hash(%$ref);
                            last SWITCH;
                    };

    /CODE/          && do {
                            warn "can't print function ref";
                            last SWITCH;
                    };

    # DEFAULT

    warn "User defined type skipped";

}
```

Voir `perlsyn/"BLOCs de base et instruction switch"` pour d'autres exemples dans ce style.

Vous devrez parfois échanger les positions de la constante et de la variable. Par exemple, disons que vous voulez tester une réponse parmi de nombreuses vous ayant été données, mais d'une façon insensible à la casse qui permette aussi les abréviations. Vous pouvez utiliser la technique suivante si les chaînes commencent toutes par des caractères différents ou si vous désirez arranger les correspondances de façon que l'une ait la précédence sur une autre, tout comme `"SEND"` a la précédence sur `"STOP"` ici :

```
chomp($answer = <>);
if    ("SEND"  =~ /^\Q$answer/i) { print "Action is send\n"  }
elsif ("STOP"  =~ /^\Q$answer/i) { print "Action is stop\n"  }
elsif ("ABORT" =~ /^\Q$answer/i) { print "Action is abort\n" }
elsif ("LIST"  =~ /^\Q$answer/i) { print "Action is list\n"  }
elsif ("EDIT"  =~ /^\Q$answer/i) { print "Action is edit\n"  }
```

Une approche totalement différente est de créer une référence de hachage ou de fonction.

```
my %commands = (
    "happy" => \&joy,
    "sad",  => \&sullen,
    "done"  => sub { die "See ya!" },
    "mad"   => \&angry,
);

print "How are you? ";
chomp($string = <STDIN>);
if ($commands{$string}) {
    $commands{$string}->();
} else {
    print "No such command: $string\n";
}
```

2.23 Comment intercepter les accès aux variables, aux fonctions, aux méthodes indéfinies ?

La méthode AUTOLOAD, dont il est question dans Autochargement in *perlsub* et AUTOLOAD : les méthodes mandataires in *perltoot*, vous permet de capturer les appels aux fonctions et aux méthodes indéfinies.

Quant aux variables indéfinies qui provoqueraient un avertissement si `use warnings` est actif, il vous suffit de transformer l'avertissement en erreur.

```
use warnings FATAL => qw(uninitialized);
```

2.24 Pourquoi une méthode incluse dans ce même fichier ne peut-elle pas être trouvée ?

Quelques raisons possibles : votre héritage se trouble, vous avez fait une faute de frappe dans le nom de la méthode, ou l'objet est d'un mauvais type. Regardez dans *perltoot* pour des détails là-dessus. Vous pourriez aussi utiliser `print ref($object)` pour déterminer par quelle classe `$object` a été consacré.

Une autre raison possible de problèmes est que vous avez utilisé la syntaxe d'objet indirect (eg, `find Guru "Samy"`) sur le nom d'une classe avant que Perl n'ait vu qu'un tel paquetage existe. Il est plus sage de s'assurer que vos paquetages sont tous définis avant de commencer à les utiliser, ce qui sera fait si vous utilisez l'instruction `use` au lieu de `require`. Sinon, assurez-vous d'utiliser la notation fléchée à la place (eg, `Guru->find("Samy")`). La notation d'objet est expliquée dans *perlobj*.

Assurez-vous de vous renseigner sur la création des modules en lisant *perlmod* et les périls des objets indirects dans Appel de méthodes in *perlobj*.

2.25 Comment déterminer mon paquetage courant ?

Si vous êtes juste un programme pris au hasard, vous pouvez faire ceci pour découvrir quel est le paquetage compilé sur le moment :

```
my $packname = __PACKAGE__;
```

Mais si vous êtes une méthode et que vous voulez afficher un message d'erreur qui inclut le genre d'objet sur lequel vous avez été appelée (qui n'est pas nécessairement le même que celui dans lequel vous avez été compilée) :

```
sub amethod {
    my $self  = shift;
    my $class = ref($self) || $self;
    warn "called me from a $class object";
}
```

2.26 Comment commenter un grand bloc de code perl ?

Vous pouvez utiliser POD pour le désactiver. Entourer le bloc à commenter de marqueurs POD. La directive `=begin` marque le début d'une section pour un formateur particulier. Utilisez le format `comment`, qu'aucun formateur ne risque de comprendre (par convention). Marquer la fin du bloc par `=end`.

```
# le programme est ici

=begin comment

tout ce truc

ici sera ignoré
par tout le monde

=end commentaire

=cut

# le programme continue
```

Les directives POD ne se placent pas n'importe où. Vous devez mettre une directive POD là où l'analyseur syntaxique attend une nouvelle instruction, pas juste au milieu d'une expression ou d'un autre élément grammaticale quelconque.

Voir *perlpod* pour plus de détails.

2.27 Comment supprimer un paquetage ?

Utilisez ce code, fourni par Mark-Jason Dominus:

```
sub scrub_package {
    no strict 'refs';
    my $pack = shift;
    die "Shouldn't delete main package"
        if $pack eq "" || $pack eq "main";
    my $stash = *{$pack . '::'}{HASH};
    my $name;
    foreach $name (keys %$stash) {
        my $fullname = $pack . '::' . $name;
        # Get rid of everything with that name.
        undef $$fullname;
        undef @$fullname;
        undef %$fullname;
        undef &$fullname;
        undef *$fullname;
    }
}
```

Ou, si vous utilisez une version récente de Perl, vous pouvez juste utiliser à la place la fonction Symbol::delete_package().

2.28 Comment utiliser une variable comme nom de variable ?

Les débutants pensent souvent qu'ils veulent qu'une variable contienne le nom d'une variable.

```
$fred    = 23;
$varname = "fred";
++$$varname;         # $fred vaut maintenant 24
```

Ceci marche *parfois*, mais c'est une très mauvaise idée pour deux raisons.

La première raison est que cela *ne marche que pour les variables globales*. Cela signifie que si $fred était une variable lexicale créée avec my() dans le code ci-dessus, alors le code ne marcherait pas du tout : vous accéderiez accidentellement à la variable globale et passeriez complètement par-dessus le lexical privé. Les variables globales sont mauvaises car elles peuvent facilement entrer en collision accidentellement et produisent en général un code non-extensible et confus.

Les références symboliques sont interdites sous le pragma `use strict`. Ce ne sont pas de vraies références et par conséquent elles ne sont pas comptées dans les références ni traitées par le ramasse-miettes.

La seconde raison qui rend mauvaise l'utilisation d'une variable pour contenir le nom d'une autre variable est que, souvent, le besoin provient d'un manque de compréhension des structures de données de Perl, en particulier des hachages. En utilisant des références symboliques, vous utilisez juste le hachage de la table de symboles du paquetage (comme `%main::`) à la place d'un hachage défini par l'utilisateur. La solution est d'utiliser à la place votre propre hachage ou une vraie référence.

```
$USER_VARS{"fred"} = 23;
$varname = "fred";
$USER_VARS{$varname}++;  # ce n'est pas $$varname++
```

Ici nous utilisons le hachage %USER_VARS à la place de références symboliques. Parfois, cela est nécessaire quand on lit des chaînes auprès de l'utilisateur avec des références de variable et qu'on veut les étendre aux valeurs des variables du programme perl. C'est aussi une mauvaise idée car cela fait entrer en conflit l'espace de noms adressable par le programme et celui adressable par l'utilisateur. Au lieu de lire une chaîne et de la développer pour obtenir le contenu des variables de votre programme :

```
$str = 'this has a $fred and $barney in it';
$str =~ s/(\$\w+)/$1/eeg;   # besoin d'un double eval
```

Il serait meilleur de conserver un hachage comme %USER_VARS et d'avoir des références de variable pointant effectivement vers des entrées de ce hachage :

```
$str =~ s/\$(\w+)/$USER_VARS{$1}/g;   # pas de /e du tout ici
```

C'est plus rapide, plus propre, et plus sûr que l'approche précédente. Bien sûr, vous n'avez pas besoin d'utiliser un dollar. Vous pourriez utiliser votre propre caractère pour rendre le tout moins confus, comme des symboles de pourcents, etc.

```
$str = 'this has a %fred% and %barney% in it';
$str =~ s/%(\w+)%/$USER_VARS{$1}/g;   # pas de /e du tout ici
```

Une autre raison qui amène les gens à croire qu'ils ont besoin qu'une variable contienne le nom d'une autre variable est qu'ils ne savent pas comment construire des structures de données correctes en utilisant des hachages. Par exemple, supposons qu'ils aient voulu deux hachages dans leur programme : %fred et %barney, et veuillent utiliser une autre variable scalaire pour s'y référer par leurs noms.

```
$name = "fred";
$$name{WIFE} = "wilma";     # définit %fred

$name = "barney";
$$name{WIFE} = "betty";     # définit %barney
```

C'est toujours une référence symbolique, avec toujours les problèmes énumérés ci-dessus sur les bras. Il serait bien meilleur d'écrire :

```
$folks{"fred"}{WIFE}   = "wilma";
$folks{"barney"}{WIFE} = "betty";
```

Et de simplement utiliser un hachage multiniveau pour commencer.

La seule occasion où vous *devez* absolument utiliser des références symboliques est lorsque vous devez réellement vous référer à la table de symboles. Cela peut se produire parce qu'il s'agit de quelque chose dont on ne peut pas prendre une vraie référence, tel qu'un nom de format. Faire cela peut aussi être important pour les appels de méthodes, puisqu'ils passent toujours par la table de symboles pour leur résolution.

Dans ces cas, vous devrez supprimer temporairement les `strict 'refs'` de façon à pouvoir jouer avec la table de symboles. Par exemple :

```
@colors = qw(red blue green yellow orange purple violet);
for my $name (@colors) {
    no strict 'refs';  # renege for the block
    *$name = sub { "<FONT COLOR='$name'>@_</FONT>" };
}
```

Toutes ces fonctions (red(), blue(), green(), etc.) apparaissent comme différentes, mais le vrai code dans la fermeture n'a effectivement été compilé qu'une fois.

Vous voudrez donc parfois utiliser les références symboliques pour manipuler directement la table de symboles. Cela n'a pas d'importance pour les formats, les handles, et les sous-programmes, car ils sont toujours globaux – vous ne pouvez pas utiliser my() sur eux. Mais pour les scalaires, les tableaux et les hachages – et habituellement pour les sous-programmes – vous préférerez probablement n'utiliser que de vraies références (les références dures).

2.29 Que signifie "bad interpreter" ?

(contribution de brian d foy)

Le message "bad interpreter" provient du shel et non de perl. Le message exact dépend de votre pleteforme, du shell utilisé et de règlage des locale.

Si vous voyez "bad interpreter - no such file or directory", c'est que la première ligne de votre script (la ligne "shebang") ne contient pas un chemin correcte vers perl (ou n'import quel programme capable de comprendre le script). Parfois cela arrive lorsque vous passez un script d'une machine à une autre dont le perl est installé différement – /usr/bin/perl au lieu de /usr/local/bin/perl par exemple. Cela peut aussi indiquer que le fichier source contient CRLF comme fin de lignes alors que la machine s'attend uniquement à LF : le shell cherche /usr/bin/perl<CR> qu'il ne trouve évidemment pas.

Si vous voyez "bad interpreter: permission denied", il vous faut rendre votre script exécutable.

Dans tous les cas, vous devriez pouvoir faire exécuter explicitement le script par perl :

```
% perl script.pl
```

Si vous obtenez un message du genre "perl: command not found", alors perl n'est pas dans votre PATH ce qui signifie que perl n'est pas là où vous l'attendez et qu'il vous faudra corriger la ligne de "shebang".

3 AUTEUR ET COPYRIGHT

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

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

4.3 Relecture

Régis Julié <*Regis.Julie@cetelem.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.