perlintro

Table des matières

1 NAME/NOM

perlintro - Brève introduction et vue d'ensemble de Perl

2 DESCRIPTION

Ce document a pour but de donner une vue d'ensemble du langage de programmation Perl tout en fournissant quelques pointeurs vers de la documentation complémentaire. Il est conçu comme un guide de démarrage destiné à un public de novices. Il apporte juste assez d'informations pour permettre de lire des programmes Perl écrits par d'autres et de comprendre approximativement ce qu'ils font, ou d'écrire des scripts simples.

Ce document ne cherche pas à être complet. Il ne tente même pas d'être parfaitement exact. Dans certains exemples la perfection a été sacrifiée dans le seul but de faire passer l'idée générale. Il vous est *fortement* recommandé de compléter cette introduction par la lecture du manuel de Perl, dont la table des matière peut être trouvée dans *perltoc*.

Tout au long de ce document vous découvrirez des références à différentes parties de la documentation Perl. Vous pouvez lire cette documentation avec la commande `perldoc` ou toute autre méthode, par exemple celle que vous utilisez pour lire le présent document.

2.1 Qu'est-ce que Perl ?

Perl est un langage de programmation généraliste créé à l'origine pour la manipulation automatique de textes et désormais utilisé dans une large gamme de tâches, dont l'administration système, le développement web, la programmation réseau, la création d'interfaces graphiques et bien plus encore.

Le langage a pour but premier d'être pratique (facile à utiliser, efficace, complet) plutôt que beau (compact, élégant, minimaliste). Ses caractéristiques principales sont sa facilité d'utilisation, le support du style de programmation impératif (à base de procédures) et du style orienté objet (OO), l'intégration de puissantes capacités de manipulation de textes, et enfin l'une des plus impressionnantes collections au monde de modules complémentaires.

Diverses définitions de Perl peuvent être trouvées dans *perl*, *perlfaq1* et sans doute à bien d'autres endroits. Nous pouvons en déduire que différents groupes d'utilisateurs voient en Perl bien des choses différentes, mais surtout que pas mal de monde considère le sujet comme suffisamment intéressant pour en parler.

2.2 Exécuter des programmes Perl

Pour exécuter un programme Perl depuis la ligne de commande Unix :

```
perl progname.pl
```

Ou bien, placez la ligne suivante comme première ligne de votre script :

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

...et exécutez le script en tapant `/dossier/contenant/le/script.pl`. Bien sur, il faut d'abord que vous l'ayez rendu exécutable, en tapant `chmod 755 script.pl` (sous Unix).

(Cette première ligne suppose que vous disposez du programme **env**. Vous pouvez plutôt y placer directement le chemin d'accès de votre exécutable perl. Par exemple `#!/usr/bin/perl`.)

Pour plus d'informations, en particulier les instructions pour d'autres plates-formes comme Windows et Mac OS, référez vous à *perlrun*.

2.3 Filet de sécurité

Perl, par défaut, est très permissif. Afin de le rendre exigeant et donc plus robuste, il est recommandé de débuter chaque programme par les lignes suivantes :

```
#!/usr/bin/perl
use strict;
use warnings;
```

Ces deux lignes supplémentaires demandent à perl de rechercher dans votre code différents problèmes courants. Chacune d'entre elles permettant la vérification de choses différentes, vous avez donc besoin de ces deux lignes. La détection d'un problème potentiel par `use strict;` arrêtera votre code immédiatement. À l'inverse, `use warnings;` ne fera qu'afficher des avertissements (comme l'option **-w** de la ligne de commande) et laissera votre code continuer. Pour en savoir plus sur ces vérifications, lisez leur documentation respective : *strict* et *warnings*.

2.4 Les bases de la syntaxe

Un programme ou un script Perl est constitué d'une suite de "phrases". Ces phrases sont simplement saisies dans le script les unes à la suite des autres dans l'ordre désiré de leur exécution, de la manière la plus directe possible. Inutile de créer une fonction `main()` ou quoi que ce soit de la sorte.

Principale différence avec les phrases françaises, une phrase Perl se termine par un point-virgule :

```
print "Salut, la terre";
```

Les commentaires débutent par un symbole dièse et vont jusqu'à la fin de la ligne.

```
# Ceci est un commentaire
```

Les espaces ou passages à la ligne sont ignorés :

```
print
    "Salut, la terre"
    ;
```

...sauf à l'intérieur des chaînes de caractères entre guillemets :

```
# La phrase suivante passe à la ligne en plein milieu du texte
print "Salut
la terre";
```

Les chaînes de caractères littérales peuvent être délimitées par des guillemets doubles ou simples (apostrophes) :

```
print "Salut, la terre";
print 'Salut, la terre';
```

Il y a cependant une différence importante entre guillemets simple et doubles. Une chaîne de caractères encadrée par des apostrophes s'affiche telle qu'elle a été saisie, tandis que dans les chaînes entre guillemets doubles Perl remplace les variables et les caractères spéciaux (par exemple les fins de lignes `\n`) par leur valeur. On parle "d'interpolation" des variables.

```
$nom = "toto";
print "Bonjour, $nom\n";     # Affiche : Bonjour, toto
print 'Bonjour, $nom\n';     # Affiche littéralement : Bonjour, $name\n
```

Les nombres n'ont pas besoin d'être encadrés par des guillemets :

```
print 42;
```

Vous pouvez utiliser ou non des parenthèses pour les arguments des fonctions selon vos goûts personnels. Elles ne sont nécessaires que de manière occasionnelle pour clarifier des problèmes de priorité d'opérateur.

```
print("Salut, la terre\n");
print "Salut, la terre\n";
```

Vous trouverez des informations plus détaillées sur la syntaxe de Perl dans *perlsyn*.

2.5 Types de variables Perl

Perl propose trois types de variables : les scalaires, les tableaux et les tables de hachage.

Scalaires

Un scalaire représente une valeur unique :

```
my $animal = "chameau";
my $reponse = 42;
```

Les scalaires peuvent être indifféremment des chaînes de caractères, des entiers, des nombres en virgules flottante, des références (voir plus loin), plus quelques valeurs spéciales. Perl procédera automatiquement aux conversions nécessaires lorsque c'est cohérent (par exemple pour transformer en nombre une chaîne de caractère représentant un nombre). Il est inutile de déclarer au préalable le type des variables mais, en revanche, vous devez déclarer vos variables lors de leur première utilisation en utilisant le mot-clé `my`. (Ceci est l'une des exigences de `use strict;`.)

Les valeurs scalaires peuvent être utilisées de différentes manières :

```
my $animal = "chameau";
print $animal;
print "L'animal est un $animal\n";
my $nombre = 42;
print "Le carré de $nombre est " . $nombre*$nombre . "\n";
```

Perl définit également la valeur scalaire `undef` (valeur d'une variable dans laquelle on n'a encore rien rangé), qu'on peut généralement considérer comme équivalente à une chaîne vide.

En Perl, en plus des variables définies dans le programme il existe un certain nombre de scalaires "magiques" qui ressemblent à des erreurs de ponctuation. Ces variables spéciales sont utilisées pour toutes sortes de choses. Elles sont documentées dans *perlvar*. Pour l'instant la seule dont vous devez vous préoccuper est la variable `$_`. On l'appelle la variable "par défaut". Elle est utilisée comme argument par défaut par quantité de fonctions Perl et modifiée implicitement par certaines structures de boucle.

```
print;     # Affiche, par défaut, le contenu de $_
```

Les tableaux

Un tableau (ou liste, il y a une différence mais dans la suite nous emploierons indifféremment les deux) représente une liste de valeurs :

```
my @animaux = ("chameau", "lama", "hibou");
my @nombres = (23, 42, 69);
my @melange   = ("chameau", 42, 1.23);
```

Le premier élément d'un tableau se trouve à la position 0. Voici comment faire pour accéder aux éléments d'un tableau :

```
print $animaux[0];              # Affiche "chameau"
print $animaux[1];              # Affiche "lama"
```

La variable spéciale `$#tableau` donne l'index du dernier élément d'un tableau (c'est à dire 1 de moins que le nombre d'éléments puisque le tableau commence à zéro) :

```
print $melange[$#melange];       # dernier élément, affiche 1.23
```

Vous pourriez être tenté d'utiliser `$#array + 1` pour connaître le nombre d'éléments contenus dans un tableau... mais il y a plus simple. Il suffit en fait d'utiliser `@array` à un emplacement ou Perl attend une valeur scalaire ("dans un contexte scalaire") et vous obtenez directement le nombre d'éléments du tableau.

```
if (@animaux < 5) { ... }
```

Les lecteurs attentifs auront sans doute remarqués que les éléments du tableau auxquels nous accédons commencent par `$`. L'idée est qu'en accédant à un élément nous demandons de retirer une seule valeur du tableau, on demande un scalaire, on obtient donc un scalaire.

Extraire simultanément plusieurs valeurs d'un tableau :

```
@animaux[0,1];                  # donne ("chameau", "lama");
@animaux[0..2];                 # donne ("chameau", "lama", "Hibou");
@animaux[1..$#animaux];         # donne tous les éléments sauf le premier
```

C'est ce qu'on appelle une tranche de tableau.

Vous pouvez réaliser quantité de choses utiles en manipulant des listes :

```
my @tri = sort @animaux; # tri
my @sansdessusdessous = reverse @nombres; # inversion
```

Comme pour les scalaire il existe quelques "tableaux magiques". Par exemple `@ARGV` (les arguments d'appel de votre script en ligne de commande) et `@_` (les arguments transmis à un sous programme). Les tableaux spéciaux sont documentés dans *perlvar*.

Les tables de hachage

Une table de hachage (ou plus brièvement "un hash") représente un ensemble de paires clé/valeur :

```
my %fruit_couleur = ("pomme", "rouge", "banane", "jaune");
```

Vous pouvez utiliser des espaces et l'opérateur => pour présenter plus joliment le code précédent. => est un synonyme de la virgule possédant quelques propriétés particulière, par exemple se passer des guillemets pour encadrer le mot qui le précède. Cela donne :

```
my %fruit_couleur = (
    pomme  => "rouge",
    banane => "jaune",
);
```

Pour accéder aux éléments d'un hash :

```
$fruit_couleur{"pomme"};           # donne "rouge"
```

Vous avez aussi la possibilité d'obtenir la liste de toutes les clés ou de toutes les valeurs du hash grâce aux fonctions `keys()` et `values()`.

```
my @fruits = keys %fruit_couleur;
my @couleurs = values %fruit_couleur;
```

Les tables de hachage n'ont pas d'ordre interne spécifique. Vous ne pouvez donc pas prévoir dans quel ordre les listes `keys()` et `values()` renverront leurs éléments (par contre vous pouvez être sur qu'il s'agit du même ordre pour les deux). Si vous désirez accéder aux éléments d'un hash de manière triée, vous avez toujours la possibilité de trier la liste `keys` et d'utiliser une boucle pour parcourir les éléments.

Vous l'aviez deviné : comme pour les scalaires et les tableaux, il existe également quelques "hashs" spéciaux. Le plus connu est `%ENV` qui contient les variables d'environnement du système. Pour tout savoir à son sujet (et sur les autres variables spéciales), regardez *perlvar*.

Les scalaires, les tableaux et les hashs sont documentés de manière plus complète dans *perldata*.

Il est possible de construire des types de données plus complexes en utilisant les références qui permettent, par exemple, de construire des listes et des hashs à l'intérieur de listes et de hashs.

Une référence est une valeur scalaire qui (comme son nom l'indique) se réfère à n'importe quel autre type de données Perl. Donc en conservant des références en tant qu'élément d'un tableau ou d'un hash, on peut facilement créer des listes et des hash à l'intérieur de listes et de hashs. Pas de panique, c'est plus compliqué à décrire qu'à créer. L'exemple suivant montre comment construire un hash à deux niveaux à l'aide de références anonymes vers des hash.

```
my $variables = {
    scalaire => {
                 description => "élément isolé",
                 prefix => '$',
                },
    tableau  => {
                 description => "liste ordonnée d'éléments",
                 prefix => '@',
                },
    hash     => {
                 description => "paire clé/valeur",
                 prefix => '%',
                },
};

print "Les scalaires commencent par $variables->{'scalaire'}->{'prefix'}\n";
```

Des informations exhaustives sur les références peuvent être trouvées dans *perlreftut*, *perllol*, *perlref* et *perldsc*.

2.6 Portée des variables

Dans tout ce qui précède, pour définir des variables nous avons utilisé sans l'expliquer la syntaxe :

```
my $var = "valeur";
```

Le mot clé `my` est en réalité optionnel. Vous pourriez vous contenter d'écrire :

```
$var = "valeur";
```

Toutefois, la seconde version va créer des variables globales connues dans tout le programme, ce qui est une mauvaise pratique de programmation. Le mot clé `my` crée des variables à portées lexicales, c'est-à-dire qui ne sont connues qu'au sein du bloc (c.-à-d. un paquet de phrases entouré par des accolades) dans lequel elles sont définies.

```
my $x = "foo";
ly $une_condition = 1m
if ($une_condition) {
    my $y = "bar";
    print $x;           # Affiche "foo"
    print $y;           # Affiche "bar"
}
print $x;               # Affiche "foo"
print $y;               # N'affiche rien; $y est hors de portée
```

De plus, en utilisant `my` en combinaison avec la phrase `use strict;` au début de vos scripts Perl, l'interpréteur détectera un certain nombre d'erreurs de programmation communes. Dans l'exemple précédant, la phrase finale `print $b;` provoquerait une erreur de compilation et vous interdirait d'exécuter le programme (ce qui est souhaitable car quand il y a une erreur il vaut mieux la détecter le plus tôt possible). Utiliser `strict` est très fortement recommandé.

2.7 Structures conditionnelles et boucles

Perl dispose des structures d'exécution conditionnelles et des boucles usuelles des autres langages de programmation, à l'exception de case/switch (mais si vous y tenez vraiment il existe un module Switch intégré à partir de Perl 5.8 et disponible sur CPAN. Lisez ci-dessous la section consacrée aux modules pour plus d'informations sur les modules et CPAN).

Une condition peut être n'importe quelle expression Perl. Elle est considérée comme vraie ou fausse suivant sa valeur, 0 ou chaîne vide signifiant FAUX et toute autre valeur signifiant VRAIE. Voyez la liste des opérateurs dans la prochaine section pour savoir quels opérateurs logiques et booléens sont habituellement utilisés dans les expressions conditionnelles.

if

```
if ( condition ) {
    ...
} elsif ( autre condition ) {
    ...
} else {
    ...
}
```

Il existe également une version négative du if :

```
unless ( condition ) {
    ...
}
```

`unless` permet ainsi de disposer d'une version plus lisible de `if (!condition)`.

Notez que, contrairement aux pratiques d'autres langages, les accolades sont obligatoires en Perl même si le bloc conditionnel est réduit à une ligne. Il existe cependant un truc permettant de donner aux expressions conditionnelles d'une ligne un aspect plus proche de l'anglais courant :

```
# comme d'habitude
if ($zippy) {
    print "Yahou!";
}
unless ($bananes) {
    print "Y'a plus de bananes";
}

# la post-condition Perlienne
print "Yahou!" if $zippy;
print "Y'a plus de bananes" unless $bananes;
```

while

```
while ( condition ) {
    ...
}
```

Il existe également une version négative de `while` :

```
until ( condition ) {
    ...
}
```

Vous pouvez aussi utiliser `while` dans une post-condition :

```
print "LA LA LA\n" while 1;          # boucle infinie
```

for

La construction `for` fonctionne exactement comme en C :

```
for ($i = 0; $i <= $max; $i++) {
    ...
}
```

La boucle `for` à la mode C est toutefois rarement nécessaire en Perl dans la mesure où Perl fournit une alternative plus intuitive : la boucle de parcours de liste `foreach`.

foreach

```
foreach (@array) {
    print "L'élément courant est $_\n";
}

print $list[$_] foreach 0 .. $max;

# vous n'êtes pas non plus obligé d'utiliser $_ ...
foreach my $cle (keys %hash) {
    print "La valeur de $cle est $hash{$cle}\n";
}
```

En pratique on peut substituer librement `for` à `foreach` et inversement. Perl se charge de détecter la variante utilisée.

Pour plus de détails sur les boucles (ainsi qu'un certain nombre de choses que nous n'avons pas mentionnées ici) consultez *perlsyn*.

2.8 Opérateurs et fonctions internes

Perl dispose d'une large gamme de fonctions internes. Nous avons déjà vu quelques unes d'entre elles dans les exemples précédents : `print`, `sort` et `reverse`. Pour avoir une liste des fonctions disponibles consultez *perlfunc*. Vous pouvez facilement accéder à la documentation de n'importe quelle fonction en utilisant `perldoc -f functionname`.

Les opérateurs Perl sont entièrement documentés dans *perlop*. Voici déjà quelques-uns parmi les plus utilisés :

Arithmétique

```
+   addition
-   soustraction
*   multiplication
/   division
```

Comparaison numérique

```
==  égalité
!=  inégalité
<   inférieur
>   supérieur
<=  inférieur ou égal
>=  supérieur ou égal
```

Comparaison de chaînes

```
eq  égalité
ne  inégalité
lt  inférieur
gt  supérieur
le  inférieur ou égal
ge  supérieur ou égal
```

Pourquoi Perl propose-t-il des opérateurs différents pour les comparaisons numériques et les comparaisons de chaînes ? Parce qu'en Perl il n'existe pas de différence de type entre variables numériques ou chaînes de caractères. Perl a donc besoin de savoir s'il faut comparer les éléments suivants dans l'ordre numérique (ou 99 est inférieur à 100) ou dans l'ordre alphabétique (où 100 vient avant 99).

Logique booléenne

`&&`	`and`	et
`||`	`or`	ou
`!`	`not`	négation

`and`, `or` et `not` ne sont pas mentionnés dans la table uniquement en tant que description des opérateurs symboliques correspondants – ils existent comme opérateurs en tant que tels. Leur raison d'être n'est pas uniquement d'offrir une meilleure lisibilité que leurs équivalent C. Ils ont surtout une priorité différente de `&&` et consorts. Lisez *perlop* pour de plus amples détails.

Divers

`=`	affectation
`.`	concaténation de chaînes
`x`	multiplication de chaînes
`..`	opérateur d'intervalle (crée une liste de nombres)

De nombreux opérateurs peuvent être combinés avec un =

```
$a += 1;        # comme $a = $a + 1
$a -= 1;        # comme $a = $a - 1
$a .= "\n";     # comme $a = $a . "\n";
```

2.9 Fichiers et E/S (entrées/sorties)

Vous pouvez ouvrir un fichier en entrée ou en sortie grâce à la fonction `open()`. Celle-ci est documentée avec un luxe extravagant de détails dans *perlfunc* et *perlopentut*. Plus brièvement :

```
open(my $in,  "<",  "input.txt")
  or die "Impossible d'ouvrir input.txt en lecture : $!";
open(my $out, ">",  "output.txt")
  or die "Impossible d'ouvrir output.txt en écriture : $!";
open(my $log, ">>", "my.log")
  or die "Impossible d'ouvrir my.log en ajout : $!";
```

Pour lire depuis un descripteur de fichier ouvert on utilise l'opérateur <>. Dans un contexte scalaire, cet opérateur lit une ligne du fichier associé. Dans un contexte de liste, il lit l'intégralité du fichier en rangeant chaque ligne dans un élément de la liste.

```
my $ligne  = <$in>;
my @lignes = <în>;
```

Lire un fichier entier en une seule fois se dit "slurper". Même si cela peut parfois s'avérer utile, c'est généralement un gâchis de mémoire. La majorité des traitements de nécessite pas de lire plus d'une ligne à la fois en utilisant les structures de boucles de Perl.

L'opérateur <> apparaît en général dans une boucle `while` :

```
while (<$in>) {  # chaque ligne est successivement affectée à $_
    print "Je viens de lire la ligne : $_";
}
```

Nous avons déjà vu comment écrire sur la sortie standard en utilisant la fonction `print()`. Celle-ci peut également prendre comme premier argument optionnel un descripteur de fichier, précisant dans quel fichier l'écriture doit avoir lieu :

```
print STDERR "Dernier avertissement.\n";
print $out $record;
print $log $logmessage;
```

Quand vous avez fini de travailler avec vos descripteurs de fichier, vous devez en principe les fermer à l'aide de la fonction `close()` (quoique pour être tout à fait honnêtes, Perl se chargera de faire le ménage si vous oubliez) :

```
close $in;
```

2.10 Expressions régulières (ou rationnelles)

Dans le jargon informatique on désigne par "expressions régulières" (ou rationnelles) une syntaxe utilisée pour définir des motifs recherchés dans un texte ou une chaîne de caractères. Le support par Perl des expressions régulières est à la fois large et puissant et c'est le sujet d'une documentation très complète : *perlrequick*, *perlretut* et autres. Nous allons les présenter brièvement :

Détection de motifs simples

```
if (/foo/)  { ... }       # vrai si $_ contient "foo"
if ($a =~ /foo/) { ... }  # vrai si $a contient "foo"
```

L'opérateur `//` de détection de motif est documenté dans *perlop*. Par défaut il travaille sur la variable `$_` ou peut être appliqué à une autre variable en utilisant l'opérateur de liaison `=~` (lui aussi documenté dans *perlop*).

Substitution simple

```
s/foo/bar/;        # remplace foo par bar dans $_
$a =~ s/foo/bar/;  # remplace foo par bar dans $a
$a =~ s/foo/bar/g; # remplace TOUTES LES INSTANCES de foo par bar dans $a
```

L'opérateur de substitution `s///` est documenté à la page *perlop*.

Expressions régulières plus complexes

Vous n'êtes pas limité à la détection de motifs fixes (si c'était le cas, on ne parlerait d'ailleurs pas d'expressions régulières, regexp pour les intimes). En pratique il est possible de détecter pratiquement n'importe quel motif imaginable en utilisant des expressions régulières plus complexes. Celles-ci sont documentées en profondeur dans *perlre*. Pour vous mettre en bouche voici déjà une petite antisèche :

```
.             un caractère unique (n'importe lequel)
\s            un blanc (espace, tabulation, passage à la ligne...)
\S            un caractère non-blanc (le contraire du précédent)
\d            un chiffre (0-9)
\D            un non-chiffre
\w            un caractère alphanumérique (a-z, A-Z, 0-9, _)
\W            un non-alphanumérique
[aeiou]       n'importe quel caractère de l'ensemble entre crochets
[^aeiou]      n'importe quel caractère sauf ceux de l'ensemble entre crochets
(foo|bar|baz) n'importe laquelle des alternatives proposées

^             le début d'une chaîne de caractères
$             la fin d'une chaîne de caractères
```

Des quantificateurs peuvent être utilisés pour indiquer combien des éléments précédents vous désirez, un élément désignant aussi bien un caractère littéral qu'un des méta-caractères énumérés plus haut, ou encore un groupe de caractères ou de méta-caractères entre parenthèses.

```
*             zéro ou plus
+             un ou plus
?             zéro ou un
{3}           exactement 3 fois l'élément précédent
{3,6}         entre 3 et 6 fois l'élément précédent
{3,}          3 ou plus des éléments précédents
```

Quelques exemples rapides :

```
/^\d+/        une chaîne commençant par un chiffre ou plus
/^$/          une chaîne vide (le début et la fin sont adjacents)
/(\d\s){3}/   un groupe de trois chiffres, chacun suivi par un blanc
              (par exemple "3 4 5 ")
/(a.)+/       une chaîne dont toutes les lettres impaires sont des a
              (par exemple "abacadaf")

# La boucle suivante lit l'entrée standard
# et affiche toutes les lignes non vides :
while (<>) {
    next if /^$/;
    print;
}
```

Capturer grâce aux parenthèses

En plus de créer un regroupement de caractères sur lequel utilisé un quantificateur, les parenthèses servent un second but. Elles peuvent être utilisées pour capturer les résultats d'une portion de regexp pour un usage ultérieur. Les résultats sont conservés dans les variables `$1`, `$2` et ainsi de suite.

```
# la méthode du pauvre pour décomposer une adresse e-mail

if ($email =~ /([^@]+)@(.+)/) {
    print "Compte : $1\n";
    print "Hôte   : $2\n";
}
```

Autres possibilités des regexp

Les regexps de Perl supportent également les références arrière, les références avant et toutes sortes d'autres constructions complexes. Pour tout savoir lisez *perlrequick*, *perlretut* et *perlre*.

2.11 Écriture de sous-programmes

Rien n'est plus facile que de déclarer un sous-programme :

```
sub logger {
    my $logmessage = shift;
    open my $logfile, ">>", "my.log" or die "Impossible d'ouvrir my.log: $!";
    print $logfile $logmessage;
}
```

Vous pouvez maintenant utiliser ce sous-programme comme n'importe quelle autre fonction prédéfinie :

```
logger("Nous avons un sous-programme de log !");
```

Que peut bien vouloir dire ce `shift` ? En réalité, comme nous l'avions évoqué plus haut, les paramètres sont transmis aux sous-programmes à travers le tableau magique `@_` (voir *perlvar* pour plus de détails). Il se trouve que la fonction `shift`, qui attend une liste, utilise `@_` comme argument par défaut. Donc la ligne `my $logmessage = shift;` extrait le premier argument de la liste et le range dans `$logmessage`.

Il existe d'autre façons de manipuler `@_` :

```
my ($logmessage, $priority) = @_;       # fréquent
my $logmessage = $_[0];                 # plus rare
```

Les sous-programmes peuvent bien entendu retourner des résultats :

```
sub square {
    my $num = shift;
    my $result = $num * $num;
    return $result;
}
```

Utilisez-le comme ceci :

```
$sq = square(8);
```

Le sujet est évidemment beaucoup plus complexe. Pour plus d'informations sur l'écriture de sous-programmes, voyez *perlsub*.

2.12 Perl orienté objet

Les possibilités objet de Perl sont relativement simples. Elles sont implémentées en utilisant des références qui connaissent le type d'objet sur lesquelles elles pointent en se basant sur le concept de paquetage de Perl. La programmation objet en Perl dépasse largement le cadre de ce document. Lisez plutôt *perlboot*, *perltoot*, *perltooc* et *perlobj*.

En tant que débutant en Perl, vous utiliserez sans doute rapidement des modules tierces-parties, dont l'utilisation est brièvement décrite ci-dessous.

2.13 Utilisations de modules Perl

Les modules Perl fournissent quantité de fonctionnalités qui vous éviteront de réinventer sans cesse la roue. Il suffit de les télécharger depuis le site CPAN (http://www.cpan.org/). Beaucoup de modules sont aussi inclus dans la distribution Perl elle-même.

Les catégories de modules vont de la manipulation de texte aux protocoles réseau, en passant par l'accès aux bases de données ou au graphisme. CPAN présente la liste des modules classés par catégorie.

Pour apprendre à installer les modules téléchargés depuis CPAN, lisez *perlmodinstall*.

Pour apprendre à utiliser un module particulier, utilisez `perldoc Module::Name`. Typiquement vous commencerez par un `use Module::Name`, qui vous donnera accès aux fonctions exportées ou à l'interface orientée objet du module.

perlfaq contient une liste de questions et les solutions à de nombreux problèmes communs en Perl et propose souvent de bons modules CPAN à utiliser.

perlmod décrit les modules Perl modules de manière plus générale. *perlmodlib* énumère les modules qui accompagnent votre installation Perl.

Si écrire des modules Perl vous démange, *perlnewmod* vous donnera d'excellents conseils.

3 AUTEUR

Kirrily "Skud" Robert <skud@cpan.org>.

L'utilisation de l'image du chameau en association avec le langage Perl est une marque déposée de O'Reilly & Associates (http://www.oreilly.com/). Utilisé avec permission.

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

Pour la traduction initiale, Christophe Grosjean <krissg@wanadoo.fr> et pour la mise à jour en version 5.10.0, Paul Gaborit (Paul.Gaborit at enstimac.fr).

4.3 Relecture

Paul Gaborit (Paul.Gaborit at enstimac.fr).

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.