perlref

Table des matières

1 NAME/NOM

perlref - Références et structures de données imbriquées en Perl

2 NOTE

Ce deocument est la documentation complète abordant tous les aspects des références. Pour une introduction n'abordant que les fonctionnalités essentielles et donc plus courte et surtout plus pédagogique, voir *perlreftut*.

3 DESCRIPTION

Avant la version 5 de Perl, il était difficile de représenter des structures de données complexes car toutes les références devaient être symboliques (et même dans ce cas, il était difficile de référencer une variable à la place d'une entrée symbolique de tableau). Desormais, non seulement Perl facilite l'utilisation de références symboliques à des variables, mais il vous laisse en plus la possibilité d'avoir des références "dures" à tout morceau de données ou de code. N'importe quel scalaire peut contenir une référence dure. Comme les tableaux et les tables de hachage contiennent des scalaires, vous pouvez désormais construire facilement des tableaux de tableaux, des tableaux de tables de hachage, des tables de hachage de tableau, des tableaux de tables de hachage de fonctions, etc.

Les références dures sont intelligentes : elles conservent la trace du nombre de références pour vous, libérant automatiquement la structure référencée quand son compteur de références atteint zéro. (Le compteur de références pour des valeurs dans des structures de données auto-référencées ou cycliques ne pourra pas atteindre zéro sans un petit coup de pouce. Cf. Ramasse-miettes à deux phases in *perlobj* pour une explication détaillée.) Si cette structure s'avère être un objet, celui-ci est détruit. Cf. *perlobj* pour de plus amples renseignements sur les objets. (Dans un certain sens, tout est objet en Perl, mais

d'habitude nous réservons ce mot pour les références à des structures qui ont été officiellement "bénies" dans un paquetage de classes.

Les références symboliques sont des noms de variables ou d'autres objets, tout comme un lien symbolique dans un système de fichiers Unix ne contient a peu de choses près que le nom d'un fichier. La notation `*glob` est un type de référence symbolique. (Les références symboliques sont parfois appelées "références douces" mais évitez de les appeler ainsi ; les références sont déjà suffisamment confuses sans ces synonymes inutiles.)

Au contraire, les références dures ressemblent plus aux liens durs dans un système de fichiers Unix : elles sont utilisées pour accéder à un objet sous-jacent sans se préoccuper de son (autre) nom. Quand le mot "référence" est utilisé sans adjectif, comme dans le paragraphe suivant, il est habituellement question d'une référence dure.

Les références sont faciles à utiliser en Perl. Il n'existe qu'un principe majeur : Perl ne référence et ne déréférence jamais de façon implicite. Quand un scalaire contient une référence, il se comporte toujours comme un simple scalaire. Il ne devient pas magiquement un tableau, une table de hachage ou une routine. Vous devez le lui préciser explicitement, en le déréférençant.

3.1 Créer des références

Les références peuvent être créées de plusieurs façons.

1. En utilisant l'opérateur "backslash" [barre oblique inverse, ndt] sur une variable, une routine ou une valeur. (Cela fonctionne plutôt comme l'opérateur & (addresse de), du C.) Notez bien que cela créer typiquement une *AUTRE* référence à la variable, parce qu'il existe déjà une telle référence dans la table des symboles. Mais même si la référence de la table des symboles disparait, vous aurez toujours la référence que la backslash a retournée. Voici quelques exemples :

   ```
   $scalarref = \$foo;
   $arrayref  = \@ARGV;
   $hashref   = \%ENV;
   $coderef   = \&handler;
   $globref   = \*foo;
   ```

   Il est impossible de créer une véritable référence à un descripteur d'E/S (descripteur de fichier ou de répertoire) en utilisant l'opérateur backslash. Le mieux que vous puissiez obtenir est une référence à un typeglob, qui est en fait une entrée complète de la table des symboles. Voir l'explication de la syntaxe `*foo{THING}` ci-dessous. Quoi qu'il en soit, vous pouvez toujours utiliser les typeglobs et les globrefs comme s'il étaient des descripteur d'E/S.

2. Une référence à un tableau anonyme peut être créée en utilisant des crochets :

   ```
   $arrayref = [1, 2, ['a', 'b', 'c']];
   ```

   Ici, nous avons créé une référence à un tableau anonyme de trois éléments, dont le dernier est lui-même une référence à un autre tableau anonyme de trois éléments. (La syntaxe multidimensionnelle décrite plus loin peut être utilisée pour y accéder. Par exemple, après le code ci-dessus, `$arrayref->[2][1]` aura la valeur "b".)
   Prendre une référence à une liste enumérée n'est pas la même chose que d'utiliser des crochets (c'est plutôt la même chose que créer une liste de références !)

   ```
   @list = (\$a, \@b, \%c);
   @list = \($a, @b, %c);      # identique !
   ```

   À l'exception de `\(@foo)` qui retourne une liste de références au contenu de `@foo`, et non pas une référence à `@foo` lui-même. Il en est de même pour `%foo` sauf évidemment pour les clés elle-mêmes qui seront simplement recopiées (puisque les clés sont justes des chaînes de caractères et non des scalaires au sens large).

3. Une référence à une table de hachage anonyme peut être créée en utilisant des accolades :

   ```
   $hashref = {
       'Adam'  => 'Eve',
       'Clyde' => 'Bonnie',
   };
   ```

   Les composants de table de hachage et de tableau comme ceux-ci peuvent être librement mélangés pour produire une structure aussi complexe que vous le souhaitez. La syntaxe multidimensionnelle décrite ci-dessous fonctionne pour ces deux cas. Les valeurs ci-dessus sont littérales mais des variables et expressions fonctionneraient de la même manière, car l'opérateur d'affectation en Perl (même à l'intérieur d'un local() ou d'un my()) sont des instructions exécutables et non pas des déclarations à la compilation.

Comme les accolades sont utilisées pour bien d'autres choses, y compris les BLOCs, vous pourriez être amené à devoir expliciter les accolades au début d'une instruction en ajoutant un `+` ou un `return` devant, de telle sorte que Perl comprenne que l'accolade ouvrante n'est pas le commencement d'un BLOC. Les économies réalisées et la valeur mnémotechnique des accolades valent bien cet embarrassement supplémentaire.

Par exemple, si vous désirez une fonction qui crée une nouvelle table de hachage et retourne une référence à celle-ci, vous avez ces possibilités :

```
sub hashem {        { @_ } }   # silencieusement faux
sub hashem {       +{ @_ } }   # correct
sub hashem { return { @_ } }   # correct
```

D'un autre côté, si vous souhaitez l'autre signification, vous pouvez faire ceci :

```
sub showem {        { @_ } }   # ambigu (correct pour le moment
                               # mais pourrait changer)
sub showem {       {; @_ } }   # correct
sub showem { { return @_ } }   # correct
```

Les `+{` et `{;` en début servent à différencier de manière explicite soit une référence à un TABLE DE HACHAGE soit un BLOC.

4. Une référence à une routine anonyme peut être créée en utilisant `sub` sans nom de routine :

```
$coderef = sub { print "Boink!\n" };
```

Notez la présence du point-virgule. À part le fait que le code à l'intérieur n'est pas exécuté immédiatement, un `sub {}` n'est ni plus ni moins qu'une déclaration comme opérateur, tout comme `do{}` ou `eval{}`. (Peu importe le nombre de fois que vous allez exécuter cette ligne particulière – à moins que vous soyez dans un `eval("...")` – $coderef fera toujours référence à la *MÊME* routine anonyme.)

Les routines anonymes fonctionnent comme les fermetures, en respectant les variables my(), c'est-à-dire les variables lexicalement visibles dans la portée actuelle. La fermeture est une notion provenant de l'univers Lisp qui indique que, si vous définissez une fonction anonyme dans un contexte lexical particulier, elle essaiera de fonctionner dans ce contexte, même quand elle est appelée en-dehors de ce contexte.

En termes plus humains, c'est une façon amusante de passer des arguments à une routine, aussi bien lorsque vous la définissez que lorsque vous l'appelez. C'est très utile pour mettre au point des petits morceaux de code à exécuter plus tard, comme les callbacks. Vous pouvez même faire de l'orienté objet avec ça bien que Perl fournisse déjà un autre mécanisme pour le faire (voir *perlobj*).

Vous pouvez aussi considérer la fermeture comme une façon d'écrire un modèle de routine sans utiliser eval(). Voici un petit exemple de fonctionnement des fermetures :

```
sub newprint {
    my $x = shift;
    return sub { my $y = shift; print "$x, $y !\n"; };
}
$h = newprint("Bonjour");
$g = newprint("Salutations");

# Un ange passe...

&$h("monde");
&$g("humains");
```

Ce qui affiche

```
Bonjour, monde !
Salutations, humains !
```

Notez en particulier que $x continue à référencer la valeur passée à newprint() *bien que* le "my $x" semble être hors de la portée au moment où la routine anonyme est exécutée. Voici donc ce qu'est la fermeture.

À propos, ceci ne s'applique qu'aux variables lexicales. Les variables dynamiques continuent de fonctionner comme elle l'ont toujours fait. La fermeture n'est pas une chose dont la plupart des programmeurs Perl ont besoin de s'embarraser pour commencer.

5. Les références sont souvent retournées par des routines spéciales appelées constructeurs. Les objets Perl sont juste des références à un type particulier d'objet qui s'avère capable de connaître quel paquetage y est associé. Les constructeurs sont juste des routines particulières qui savent comment créer cette association. Ils le font en commençant par une référence ordinaire qui reste telle quelle même si c'est un objet. Les constructeurs sont souvent nommés new() et appelés indirectement :

```
$objref = new Doggie (Tail => 'short', Ears => 'long');
```

Mais il n'est pas nécessaire d'avoir :

```
$objref   = Doggie->new(Tail => 'short', Ears => 'long');

use Term::Cap;
$terminal = Term::Cap->Tgetent( { OSPEED => 9600 });

use Tk;
$main    = MainWindow->new();
$menubar = $main->Frame(-relief              => "raised",
                        -borderwidth         => 2)
```

6. Des références de type approprié peuvent venir a exister si vous les déréférencez dans un contexte qui suppose qu'elles existent. Comme nous n'avons pas encore parlé du déréférencement, nous ne pouvons toujorus pas vous montrer d'exemples.

7. Une référence peut être créée en utilisant une syntaxe particulière, sentimentalement connue comme la syntaxe *foo{THING}. *foo{THING} retourne une référence à l'emplacement THING dans *foo (qui est l'entrée de la table des symboles contenant tout ce qui est connu en tant que "foo").

```
$scalarref = *foo{SCALAR};
$arrayref  = *ARGV{ARRAY};
$hashref   = *ENV{HASH};
$coderef   = *handler{CODE};
$ioref     = *STDIN{IO};
$globref   = *foo{GLOB};
$formatref = *foo{FORMAT};
```

Tout ceci s'explique de lui-même, à part `*foo{IO}`. Il retourne le descripteur d'E/S utilisé pour les descripteurs de fichiers (open in *perlfunc*), de sockets (socket in *perlfunc* et socketpair in *perlfunc*) et de répertoires (opendir in *perlfunc*). Pour des raisons de compatibilités avec les versions précédentes de Perl, *foo{FILEHANDLE} est un synonyme de *foo{IO}. Si les avertissements sont actifs, l'utilisation de ce synonyme affichera un message.

`*foo{TURC}` retourne un indéfini si ce TRUC particulier n'a pas été utilisé auparavant, sauf dans le cas des scalaires. *foo{SCALAR} retourne une référence à un scalaire anonyme si $foo n'a pas encore été utilisé. Ceci pourrait changer dans une prochaine version.

`*foo{IO}` est une autre manière d'accéder au mécanisme *HANDLE indiqué dans Typeglobs et Handles de Fichiers in *perldata* pour passer des descripteurs de fichiers comme arguments ou comme valeur de retour de routines, ou pour les stocker dans des structures de données plus grandes. L'inconvénient, c'est qu'il ne crée pas de nouveau descripteur de fichier pour vous. L'avantage, c'est qu'il y a moins de risque d'aller au-delà de ce que vous souhaitez qu'avec une affection de typeglob (il passe tout de même les descripteurs de fichier et de répertoire). Ceci étant, si vous l'affectez à un scalaire au lieu d'un typeglob comme dans l'exemple ci-dessous, vous êtes couvert dans tous les cas.

```
splutter(*STDOUT);     # passe tout le glob
splutter(*STDOUT{IO}); # ne passe que le descripteur
                       # de fichier et de répertoire

sub splutter {
    my $fh = shift;
    print $fh "her um well a hmmm\n";
}

$rec = get_rec(*STDIN);     # passe tout le glob
$rec = get_rec(*STDIN{IO}); # ne passe que le descripteur
                            # de fichier et de répertoire

sub get_rec {
    my $fh = shift;
    return scalar <$fh>;
}
```

3.2 Utiliser des références

C'est tout pour la création de références. Maintenant, vous devez sûrement mourir d'envie de savoir comment utiliser ces références pour en revenir à vos données perdues depuis longtemps. Il existe plusieurs méthodes de base.

1. Où que vous mettiez un identifiant (ou une chaîne d'identifiants) comme partie d'une variable ou d'une nom de routine, vous pouvez remplacer cet identifiant par une simple variable scalaire contenant une référence de type correct :

   ```
   $bar = $$scalarref;
   push(@$arrayref, $filename);
   $$arrayref[0] = "January";
   $$hashref{"KEY"} = "VALUE";
   &$coderef(1,2,3);
   print $globref "output\n";
   ```

   Il est important de comprendre qu'ici, nous ne déréférençons *pas* en particulier `$arrayref[0]` ou `$hashref{"KEY"}`. Le déréférencement de la variable scalaire a lieu *avant* toute recherche de clé. Tout ce qui est plus complexe qu'une simple variable scalaire doit utiliser les méthodes 2 et 3 ci-dessous. un "simple scalaire" inclut toutefois un identifiant qui utilise lui-même la méthode 1 de façon récursive. Le code suivant imprime par conséquent "howdy".

   ```
   $refrefref = \\\"howdy";
   print $$$$refrefref;
   ```

2. Où que vous mettiez un identifiant (ou une chaîne d'identifiants) comme partie d'une variable ou d'une nom de routine, vous pouvez remplacer cet identifiant par un BLOC retournant une référence de type correct. En d'autres mots, les exemples précédents auraient pu être écrits ainsi :

   ```
   $bar = ${$scalarref};
   push(@{$arrayref}, $filename);
   ${$arrayref}[0] = "January";
   ${$hashref}{"KEY"} = "VALUE";
   &{$coderef}(1,2,3);
   $globref->print("output\n");  # ssi IO::Handle est chargé
   ```

   Il est vrai que c'est un peu idiot d'utiliser des accolades dans ce cas-là, mais le BLOC peut contenir n'importe quelle expression, en particulier une expression subscript telle que celle-ci :

   ```
   &{ $dispatch{$index} }(1,2,3);      # appel la routine correcte
   ```

   Comme il est possible d'omettre les accolades dans le cas simple de `$$x`, les gens font souvent l'erreur de considérer le déréférencement comme des opérateurs propres et se posent des questions à propos de leur précédence. Mais s'ils en étaient, vous pourriez utiliser des parenthèses à la place des accolades. Ce qui n'est pas le cas. Remarquez la différence ci-dessous. Le cas 0 est un raccourci du cas 1 mais *pas* du cas 2.

   ```
   $$hashref{"KEY"}   = "VALUE";       # CAS 0
   ${$hashref}{"KEY"} = "VALUE";       # CAS 1
   ${$hashref{"KEY"}} = "VALUE";       # CAS 2
   ${$hashref->{"KEY"}} = "VALUE";     # CAS 3
   ```

   Le cas 2 est aussi décevant dans le sens que vous accédez à une variable appelée %hashref, sans déréférencer par $hashref la table de hachage qu'il référence probablement. Ceci correspond au cas 3.

3. Les appels de routines et les recherches d'éléments individuels de tableaux sont tellement courants qu'il devient pénible d'utiliser la méthode 2. En forme de sucre syntaxique, les exemples de la méthode 2 peuvent être écrits ainsi :

   ```
   $arrayref->[0] = "January";   # Élément de tableau
   $hashref->{"KEY"} = "VALUE";  # Élément de table de hachage
   $coderef->(1,2,3);            # Appel d'une routine
   ```

   La partie gauche de la flèche peut être n'importe quelle expression retournant une référence, y compris un déréférencement précédent. Notez que `$array[$x]` n'est *pas* ici la même chose que `$array->[$x]` :

   ```
   $array[$x]->{"foo"}->[0] = "January";
   ```

C'est un des cas que nous avons mentionnés plus tôt et dans lequel les références peuvent venir à exister dans un contexte lvalue. Avant cette instruction, `$array[$x]` peut avoir été indéfini. Dans ce cas, il est automatiquement défini avec une référence de table de hachage, de telle sorte que nous puissions rechercher `{"foo"}` dedans. De la même manière, `$array[$x]->{"foo"}` sera automatiquement défini avec une référence de tableau, de telle façon que nous puissions rechercher dedans. Ce processus est appelé *autovivification*.

Encore une petite chose ici : la flèche est facultative *entre* les crochets subscripts. Vous pouvez donc abbréger le code ci-dessus en :

```
$array[$x]{"foo"}[0] = "January";
```

Ce qui, dans le cas dégénéré de la seule utilisation de tableaux ordinaires, vous donne des tableaux multidimensionnels tout comme en C :

```
$score[$x][$y][$z] += 42;
```

Bon, d'accord, pas complètement comme en C, en fait. Le C ne sait pas comment aggrandir ses tableaux à la demande. Perl le sait.

4. Si une référence se révèle être une référence à un objet, il existe alors probablement des méthodes pour accéder aux choses référencées, et vous devriez vous cantonner à ces méthodes à moins que vous ne soyez dans le paquetage de classes qui définit justement les méthodes de cet objet. En d'autres termes, soyez sages et ne violez pas l'encapsulation des objets sans d'excellentes raisons. Perl ne renforce pas l'encapsulation. Nous ne sommes pas totalitaires. En renvanche, nous attendons un minimum de politesse.

L'utilisation d'un nombre ou d'une chaîne en tant que référence en fait une référence symbolique comme expliqué plus haut. L'utilisation d'une référence en tant que nombre la transforme en un entier représentant son emplacement en mémoire. Le seul usage intéressant est la comparaison numérique de deux références pour savoir si elles se réfèrent au même emplacement.

```
if ($ref1 == $ref2) {
  print "refs 1 et 2 font référence à la même chose\n";
}
```

L'utilisation d'une référence en tant que chaîne produit à la fois le type d'objet qu'elle référence en incluant le nom du paquetage l'ayant éventuellement consacré (par bless()) comme expliqué dans *perlobj*, et aussi son adresse mémoire numérique en hexadécimal. L'opérateur ref() produit juste le type d'objet lié à la référence, sans l'adresse. Voir ref in *perlfunc* pour plus de détails et des exemples d'utilisation.

L'opérateur bless() peut être utilisé pour associer l'objet, sur lequel pointe une référence, avec un paquetage fonctionnant comme une classe d'objets. Cf. *perlobj*.

Un typeglob peut être déréférencé de la même façon qu'une référence, car la syntaxe de déréférencement indique toujours le type de référence souhaité. Par conséquent, `${*foo}` et `${\$foo}` indique tous les deux la même variable scalaire.

Voici un truc pour interpoler l'appel d'une routine dans une chaine de caractères :

```
print "My sub returned @{[mysub(1,2,3)]} that time.\n";
```

La façon dont ça marche, c'est que lorsque le `@{...}` est aperçu à l'intérieur des guillemets de la chaîne de caractères, il est évalué comme un bloc. Le bloc crée une référence à une tableau anonyme contenant le résultat de l'appel à `mysub(1,2,3)`. Le bloc entier retourne ainsi une référence à un tableau, qui est alors déréférencé par `@{...}` et inséré dans la chaîne de caractères entre guillemets. Cette chipotterie est aussi utile pour des expressions arbitraires :

```
print "That yields @{[$n + 5]} widgets\n";
```

3.3 Références symboliques

Nous avons déjà expliqué que, quand c'est nécessaire, les références devenaient existantes si elles sont définies, mais nous n'avons pas dit ce qui arrivait lorsqu'une valeur utilisée comme référence est déjà définie mais n'est *pas* une référence dure. Si vous l'utilisez comme référence dans ce cas-là, elle sera traitée comme une référence symbolique. C'est-à-dire que la valeur du scalaire est considérée comme le *nom* d'une variable, plutôt que comme un lien direct vers une (éventuelle) valeur anonyme.

En général, les gens s'attendent à ce que ça fonctionne de cette façon. C'est donc comme ça que ça marche.

```
$name = "foo";
$$name = 1;                 # Affecte $foo
${$name} = 2;               # Affecte $foo
${$name x 2} = 3;           # Affecte $foofoo
$name->[0] = 4;             # Affecte $foo[0]
@$name = ();                # Efface @foo
&$name();                   # Appelle &foo() (comme en Perl 4)
$pack = "THAT";
${"${pack}::$name"} = 5;    # Affecte $THAT::foo sans évaluation
```

C'est très puissant, et potentiellement dangereux, dans le sens où il est possible de vouloir (avec la plus grande sincérité) utiliser une référence dure, et utiliser accidentellement une référence symbolique à la place. Pour vous en prémunir, vous pouvez utiliser

```
use strict 'refs';
```

et seules les références dures seront alors autorisées dans le reste du bloc l'incluant. Un bloc imbriqué peut inverser son effet avec

```
no strict 'refs';
```

Seuls les variables (globales, même si elles sont localisées) de paquetage sont visibles par des références symboliques. Les variables lexicales (déclarées avec my()) ne font pas partie de la table des symboles, et sont donc invisibles à ce mécanisme. Par exemple :

```
local $value = 10;
$ref = "value";
{
    my $value = 20;
    print $$ref;
}
```

Ceci imprimera 10 et non pas 20. Souvenez-vous que local() affecte les variables de paquetage, qui sont toutes "globales" au paquetage.

3.4 Références pas-si-symboliques-que-ça

Une nouvelle fonctionnalité contribuant à la lisibilité en perl version 5.001 est que les crochets autour d'une référence symbolique se comportent comme des apostrophes, tout comme elles l'ont toujours été dans une chaîne de caractères. C'est-à-dire que

```
$push = "pop on ";
print "${push}over";
```

a toujours imprimé "pop on over", même si push est un mot réservé. Ceci a été généralisé pour fonctionner de même en dehors de guillemets, de telle sorte que

```
print ${push} . "over";
```

et même

```
print ${ push } . "over";
```

auront un effet identique. (Ceci aurait provoqué une erreur syntaxique en Perl 5.000, bien que Perl 4 l'autorisait dans une forme sans espaces.) Cette construction n'est *pas* considérée comme une référence symbolique lorsque vous utilisez strict refs :

```
use strict 'refs';
${ bareword };      # Correct, signifie $bareword.
${ "bareword" };    # Erreur, référence symbolique.
```

De façon similaire, à cause de tout le subscripting qui est effectué en utilisant des mots simples, nous avons appliqué la même règle à tout mot simple qui soit utilisé pour le subscripting d'une table de hachage. Désormais, au lieu d'écrire

```
$array{ "aaa" }{ "bbb" }{ "ccc" }
```

vous pourrez donc juste écrire

```
$array{ aaa }{ bbb }{ ccc }
```

sans vous inquiéter du fait que les subscripts soient ou non des mots réservés. Dans les rares cas où vous souhaiteriez faire quelque chose comme :

```
$array{ shift }
```

vous pouvez en forcer l'interprétation comme un mot réservé en ajoutant n'importe quoi qui soit plus qu'un mot simple :

```
$array{ shift() }
$array{ +shift }
$array{ shift @_ }
```

La directive `use warnings` ou l'option **-w** vous avertira si un mot réservé est interprété comme une chaîne de caractères. Mais il ne vous avertira plus si vous utilisez des mots en minuscules, car la chaîne de caractères est entre guillemets de façon effective.

3.5 Pseudo-tables de hachage : utiliser un tableau comme table de hachage

AVERTISSEMENT : cette section traite de fonctionnalités expérimentales. Certains détails pourraient changer sans annonce particulière dans les prochaines versions.

NOTE : la partie visible de l'implémentation actuelle des pseudo-tables de hachage (l'utilisation singulière du premier élément du tableau) est dépréciée à partir de Perl 5.8.0 et disparaîtra dans Perl 5.10.0. Cette fonctionnalité sera implémentée autrement. Au-delà de l'interface actuelle qui est particulièrement horrible, l'implémentation actuelle ralentit notablement l'utilisation normale des tableaux et des tables de hachage. La directive 'fields' restera disponible.

Avec la version 5.005 de Perl, vous pouvez désormais utiliser une référence à un tableau dans un contexte qui exigerait normalement une référence à une table de hachage. Ceci vous permet d'accéder aux éléments d'un tableau en utilisant des noms symboliques, comme s'ils étaient les champs d'une structure.

Pour que cela fonctionne, le tableau doit contenir des informations supplémentaires. Le premier élément du tableau doit être une référence à une table de hachage qui associe les noms de champs avec les indices du tableau. Voici un exemple :

```
$struct = [{foo => 1, bar => 2}, "FOO", "BAR"];

$struct->{foo};  # identique à $struct->[1], c'est-à-dire "FOO"
$struct->{bar};  # identique à $struct->[2], c'est-à-dire "BAR"

keys %$struct;   # retournera ("foo", "bar") dans un certain ordre
values %$struct; # retournera ("FOO", "BAR") dans le meme certain ordre

while (my($k,$v) = each %$struct) {
    print "$k => $v\n";
}
```

Perl déclenchera une exception si vous essayez d'accéder à des champs inexistants. Pour éviter les incohérences, utilisez toujours la fonction fields::phash() fournie par la directive `fields`.

```
use fields;
$pseudohash = fields::phash(foo => "FOO", bar => "BAR");
```

Pour de meilleures performances, Perl peut aussi effectuer la traduction des noms de champs en indices de tableau lors de la compilation pour les références à des objets typées. Voir *fields*.

Il existe deux moyens de vérifier l'existence d'une clé dans une pseudo-table de hachage. Le premier est d'utiliser exist(). Cela teste si ce champ donné a déjà été utilisé. Ce comportement est le même que celui d'une table de hachage normale. Par exemple :

```
use fields;
$phash = fields::phash([qw(foo bar pants)], ['FOO']);
$phash->{pants} = undef;

print exists $phash->{foo};    # vrai, 'foo' est valué dans la declaration
print exists $phash->{bar};    # faux, 'bar' n'a jamais été utilisé
print exists $phash->{pants};  # vrai, 'pants' a été utilisé
```

Le second moyen est d'utiliser exists() sur la table de hachage présente comme premier élément du tableau. Cela teste si ce champ est un champ valide pour cette pseudo-table de hachage.

```
print exists $phash->[0]{bar};   # vrai, 'bar' est valide
print exists $phash->[0]{shoes}; # faux, 'shoes' ne peut être utilisé
```

Un appel à delete() sur un élément d'une pseudo-table de hachage n'efface que le valeur correspondant à cette clé et non la clé elle-même. Pour effacer la clé, vous devez l'effacer explicitement de la table située au premier élément du tableau.

```
print delete $phash->{foo};     # affiche $phash->[1], "FOO"
print exists $phash->{foo};     # faux
print exists $phash->[0]{foo};  # vrai, la clé existe encore
print delete $phash->[0]{foo};  # maintenant la clé est effacée
print $phash->{foo};            # exception déclenchée
```

3.6 Modèles de fonctions

Comme expliqué ci-dessus, une fermeture est une fonction anonyme qui a accès aux variables lexicales qui étaient visibles lors de sa compilation. Elle conserve l'accès à ses variables même si elle n'est exécutée que plus tard, comme dans le cas des signaux ou des callbacks en Tk.

Utiliser une fermeture comme modèle de fonction nous permet de générer de nombreuses fonctions agissant de façon similaire. Supposons que vous souhaitiez des fonctions nommées d'après la couleur qu'elles produiront en HTML via la balise FONT :

```
print "Be ", red("careful"), "with that ", green("light");
```

Les fonctions red() et green() seront très similaires. Pour les créer, nous allons assigner une fermeture à un typeglob du nom de la fonction que nous voulonsq construire.

```
@colors = qw(red blue green yellow orange purple violet);
for my $name (@colors) {
  no strict 'refs'; # Autorise la manipulation de la table de symboles
  *$name = *{uc $name} = sub { "<FONT COLOR='$name'>@_</FONT>" };
}
```

Désormais, toutes ces fonctions existent de façon indépendante. Vous pouvez appeler red(), RED(), blue(), BLUE(), green(), etc. Cette technique optimise le temps de compilation et l'utilisation de la mémoire, et elle est aussi moins sujette aux erreurs puisque la vérification syntaxique a lieu à la compilation. Il est nécessaire qu'aucune variable de la routine anonyme ne soit lexicale pour créer une fermeture propre. C'est la raison pour laquelle nous avons un `my` dans notre boucle.

C'est l'un des seuls endroits où donner un prototype à une fermeture a un réel sens. Si vous souhaitiez imposer un contexte scalaire aux arguments de ces fonctions (ce qui n'est probablement pas une bonne idée pour cet exemple particulier), vous auriez pu écrire à la place :

```
  *$name = sub ($) { "<FONT COLOR='$name'>$_[0]</FONT>" };
```

Quoi qu'il en soit, comme la vérification des prototypes a lieu à la compilation, l'affectation ci-dessus est effectuée trop tard pour être vraiment utile. Vous pourriez gérer ça en insérant la boucle entière d'affectations dans un bloc BEGIN, forçant ainsi son exécution pendant la compilation.

L'accès aux lexicaux qui change au-delà des types (comme ceux de la boucle `for` ci-dessus) ne fonctionne qu'avec des fermetures et pas avec des routines générales. Par conséquent, dans le cas général, les routines nommées ne s'imbriquent pas proprement, au contraire des routines anonymes. C'est comme cela parce que les routines nommées sont crées (et récupèrent les lexicaux externes) une seule fois lors de la compilation alors que les routines anonymes réalisent cette récupération à chaque exécution de l'opérateur 'sub'. Si vous êtes habitué à l'utilisation de routines imbriquées dans d'autres langages de programmation, avec leurs propres variables privées, il va vous falloir travailler là-dessus en Perl un tant soit peu. La programmation intuitive de ce genre de choses implique des avertissements mystérieux du genre "will not stay shared". Par exemple, ceci ne fonctionnera pas :

```
sub outer {
    my $x = $_[0] + 35;
    sub inner { return $x * 19 }   # FAUX
    return $x + inner();
}
```

Une solution pourraît être celle-ci :

```
sub outer {
    my $x = $_[0] + 35;
    local *inner = sub { return $x * 19 };
    return $x + inner();
}
```

Maintenant, inner() ne peut être appelée que de l'intérieur de outer(), grâce aux affectations temporaires de la fermeture (routine anonyme). Mais lorsque cela a lieu, elle a un accès normal à la variable lexicale $x dans la portée de outer().

Ceci a pour effet intéressant de créer une fonction locale à une autre, ce qui n'est pas normalement supporté par Perl.

4 AVERTISSEMENT

Vous ne devriez pas (utilement) utiliser une référence comme clé d'une table de hachage. Elle sera convertie en chaîne de caractères :

```
$x{ \$a } = $a;
```

Si vous essayez de déréférencer la clé, il n'y aura pas de déréférencement dur et vous ne ferez pas ce que vous souhaitiez. Vous devriez plutôt faire ainsi :

```
$r = \@a;
$x{ $r } = $r;
```

Et alors, au moins, vous pourrez utiliser les valeurs, par values(), qui seront de véritables références, au contraire des clés, par keys().

Le module standard Tie::RefHash fournit une base de travail pratique pour faire ce genre de choses.

5 VOIR AUSSI

À côté de la documentation standard, du code source peut être instructif. Quelques exemples pathologiques de l'utilisation de références peuvent être trouvées dans le test de régression *t/op/ref.t* du répertoire source de Perl.

Voir aussi *perldsc* et *perllol*, pour l'utilisation de références dans la création de structures de données complexes, et *perltoot*, *perlobj* et *perlbot* pour leur utilisation dans la création d'objets.

6 AUTEUR

Larry Wall

7 TRADUCTION

7.1 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/.

7.2 Traducteur

Traduction initiale : Jean-Pascal Peltier <jp_peltier@altavista.net>. Mise à jour : Paul Gaborit <paul.gaborit at enstimac.fr>.

7.3 Relecture

Personne pour l'instant.

8 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.