

# perldsc

## Table des matières

<b>1</b>	<b>NAME/NOM</b>	<b>2</b>
<b>2</b>	<b>DESCRIPTION</b>	<b>2</b>
<b>3</b>	<b>RÉFÉRENCES</b>	<b>3</b>
<b>4</b>	<b>ERREURS COURANTES</b>	<b>3</b>
<b>5</b>	<b>AVERTISSEMENT SUR LA PRÉCÉDENCE</b>	<b>5</b>
<b>6</b>	<b>POURQUOI TOUJOURS UTILISER <code>use strict</code></b>	<b>5</b>
<b>7</b>	<b>DÉBOGAGE</b>	<b>6</b>
<b>8</b>	<b>EXEMPLES DE CODE</b>	<b>6</b>
<b>9</b>	<b>TABLEAUX DE TABLEAUX</b>	<b>6</b>
9.1	Déclaration d'un TABLEAU DE TABLEAUX . . . . .	6
9.2	Génération d'un TABLEAU DE TABLEAUX . . . . .	6
9.3	Accès et affichage d'un TABLEAU DE TABLEAUX . . . . .	7
<b>10</b>	<b>HACHAGE DE TABLEAUX</b>	<b>7</b>
10.1	Déclaration d'un HACHAGE DE TABLEAUX . . . . .	7
10.2	Génération d'un HACHAGE DE TABLEAUX . . . . .	7
10.3	Accès et affichage d'un HACHAGE DE TABLEAUX . . . . .	8
<b>11</b>	<b>TABLEAUX DE HACHAGES</b>	<b>8</b>
11.1	Déclaration d'un TABLEAU DE HACHAGES . . . . .	8
11.2	Génération d'un TABLEAU DE HACHAGES . . . . .	9
11.3	Accès et affichage d'un TABLEAU DE HACHAGES . . . . .	9
<b>12</b>	<b>HACHAGES DE HACHAGES</b>	<b>10</b>
12.1	Déclaration d'un HACHAGE DE HACHAGES . . . . .	10
12.2	Génération d'un HACHAGE DE HACHAGES . . . . .	10
12.3	Accès et affichage d'un HACHAGE DE HACHAGES . . . . .	11
<b>13</b>	<b>ENREGISTREMENTS PLUS ÉLABORÉS</b>	<b>12</b>
13.1	Déclaration d'ENREGISTREMENTS PLUS ÉLABORÉS . . . . .	12
13.2	Déclaration d'un HACHAGE D'ENREGISTREMENTS COMPLEXES . . . . .	12
13.3	Génération d'un HACHAGE D'ENREGISTREMENTS COMPLEXES . . . . .	13
<b>14</b>	<b>Liens avec les bases de données</b>	<b>14</b>
<b>15</b>	<b>VOIR AUSSI</b>	<b>14</b>
<b>16</b>	<b>AUTEUR</b>	<b>14</b>

<b>17 TRADUCTION</b>	<b>14</b>
17.1 Version	14
17.2 Traducteur	15
17.3 Relecture	15
<b>18 À propos de ce document</b>	<b>15</b>

## 1 NAME/NOM

perldsc - Livre de recettes des structures de données en Perl

## 2 DESCRIPTION

La seule caractéristique manquant cruellement au langage de programmation Perl avant sa version 5.0 était les structures de données complexes. Même sans support direct par le langage, certains programmeurs vaillants parvinrent à les émuler mais c'était un dur travail, à déconseiller aux âmes sensibles. Vous pouviez occasionnellement vous en sortir avec la notation `$m{$AoA, $b}` empruntée à *awk* dans laquelle les clés sont en fait une seule chaîne concaténée "`$AoA$b`", mais leurs parcours et leurs tris étaient difficiles. Des programmeurs un peu plus désespérés ont même directement bidouillé la table de symboles interne de Perl, une stratégie qui s'est montrée dure à développer et à maintenir - c'est le moins que l'on puisse dire.

La version 5.0 de Perl met à notre disposition des structures de données complexes. Vous pouvez maintenant écrire quelque chose comme ce qui suit pour obtenir d'un seul coup un tableau à trois dimensions !

```
for $x (1 .. 10) {
  for $y (1 .. 10) {
    for $z (1 .. 10) {
      $AoA[$x][$y][$z] =
        $x ** $y + $z;
    }
  }
}
```

À première vue cette construction apparaît simple, mais elle est en fait beaucoup plus compliquée qu'elle ne le laisse paraître !

Comment pouvez-vous l'imprimer ? Pourquoi ne pouvez-vous pas juste dire `print @AoA` ? Comment la triezy-vous ? Comment pouvez-vous la passer à une fonction ou en récupérer une depuis une fonction ? Est-ce un objet ? Pouvez-vous la sauver sur disque pour la relire plus tard ? Comment accédez-vous à des lignes ou à des colonnes entières de cette matrice ? Est-ce que toutes les valeurs doivent être numériques ?

Comme vous le voyez, il est assez facile d'être déconcerté. Ces difficultés viennent, pour partie, de l'implémentation basée sur les références mais aussi du manque de documentation proposant des exemples destinés au débutant.

Ce document est conçu pour traiter, en détail mais de façon compréhensible, toute une panoplie de structures de données que vous pourriez être amené à développer. Il devrait aussi servir de livre de recettes donnant des exemples. De cette façon, lorsque vous avez besoin de créer l'une de ces structures de données complexes, vous pouvez simplement piquer, chaparder ou dérober un exemple de ce guide.

Jetons un oeil en détail à chacune de ces constructions possibles. Il existe des sections séparées pour chacun des cas suivants :

- tableaux de tableaux
- hachages de tableaux
- tableaux de hachages
- hachages de hachages
- constructions plus élaborées

Mais pour le moment, intéressons-nous aux problèmes communs à tous ces types de structures de données.

### 3 RÉFÉRENCES

La chose la plus importante à comprendre au sujet de toutes les structures de données en Perl - y compris les tableaux multidimensionnels - est que même s'ils peuvent paraître différents, les @TABLEAUX et les %HACHAGES en Perl sont tous unidimensionnels en interne. Ils ne peuvent contenir que des valeurs scalaires (c'est-à-dire une chaîne, un nombre ou une référence). Ils ne peuvent pas contenir directement d'autres tableaux ou hachages, mais ils contiennent à la place des *références* à des tableaux ou des hachages.

Vous ne pouvez pas utiliser une référence à un tableau ou à un hachage exactement de la même manière que vous le feriez d'un vrai tableau ou d'un vrai hachage. Pour les programmeurs C ou C++, peu habitués à distinguer les tableaux et les pointeurs vers des tableaux, ceci peut être déconcertant. Si c'est le cas, considérez simplement la différence entre une structure et un pointeur vers une structure.

Vous pouvez (et devriez) en lire plus au sujet des références dans le manuel *perlref*. Brièvement, les références ressemblent assez à des pointeurs sachant vers quoi ils pointent (les objets sont aussi une sorte de référence, mais nous n'en aurons pas besoin pour le moment). Ceci signifie que lorsque vous avez quelque chose qui vous semble être un accès à un tableau ou à un hachage à deux dimensions ou plus, ce qui se passe réellement est que le type de base est seulement une entité unidimensionnelle qui contient des références vers le niveau suivant. Vous pouvez juste l'*utiliser* comme si c'était une entité à deux dimensions. C'est aussi en fait la façon dont fonctionnent presque tous les tableaux multidimensionnels en C.

```
$array[7][12]           # tableau de tableaux
$array[7]{string}      # tableau de hachages
$hash{string}[7]      # hachage de tableaux
$hash{string}{'another string'} # hachage de hachages
```

Maintenant, puisque le niveau supérieur ne contient que des références, si vous essayez d'imprimer votre tableau avec une simple fonction `print()`, vous obtiendrez quelque chose qui n'est pas très joli, comme ceci :

```
@AoA = ( [2, 3], [4, 5, 7], [0] );
print $AoA[1][2];
7
print @AoA;
ARRAY(0x83c38)ARRAY(0x8b194)ARRAY(0x8b1d0)
```

C'est parce que Perl ne déréférence (presque) jamais implicitement vos variables. Si vous voulez atteindre la chose à laquelle se réfère une référence, alors vous devez le faire vous-même soit en utilisant des préfixes d'indication de type, comme `$$blah`, `@{ $blah }`, `@{ $blah[$i] }`, soit des flèches de pointage postfixes, comme `$a->[3]`, `$h->{fred}`, ou même `$ob->method()->[3]`.

### 4 ERREURS COURANTES

Les deux erreurs les plus communes faites lors de l'affectation d'une donnée de type tableau de tableaux est soit l'affectation du nombre d'éléments du tableau (NdT : au lieu du tableau lui-même bien sûr), soit l'affectation répétée d'une même référence mémoire. Voici le cas où vous obtenez juste le nombre d'éléments au lieu d'un tableau imbriqué :

```
for $i (1..10) {
    @list = somefunc($i);
    $AoA[$i] = @list;      # FAUX !
}
```

C'est le cas simple où l'on affecte un tableau à un scalaire et où l'on obtient le nombre de ses éléments. Si c'est vraiment bien ce que vous désirez, alors vous pourriez être un poil plus explicite à ce sujet, comme ceci :

```
for $i (1..10) {
    @array = somefunc($i);
    $counts[$i] = scalar @array;
}
```

Voici le cas où vous vous référez encore et encore au même emplacement mémoire :

```

for $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = \@array;    # FAUX !
}

```

Allons bon, quel est le gros problème ? Cela semble bon, n'est-ce pas ? Après tout, je viens de vous dire que vous aviez besoin d'un tableau de références, alors flûte, vous m'en avez créé un !

Malheureusement, bien que cela soit vrai, cela ne marche pas. Toutes les références dans @AoA se réfère au *même endroit*, et elles contiendront toutes par conséquent ce qui se trouvait en dernier dans @array ! Le problème est similaire à celui du programme C suivant :

```

#include <pwd.h>
main() {
    struct passwd *getpwnam(), *rp, *dp;
    rp = getpwnam("root");
    dp = getpwnam("daemon");

    printf("daemon name is %s\nroot name is %s\n",
           dp->pw_name, rp->pw_name);
}

```

Ce qui affichera :

```

daemon name is daemon
root name is daemon

```

Le problème est que `rp` et `dp` sont des pointeurs vers le même emplacement en mémoire ! En C, vous devez utiliser `malloc()` pour vous réserver de la mémoire. En Perl, vous devez utiliser à la place le constructeur de tableau `[]` ou le constructeur de hachages `{}`. Voici la bonne façon de procéder :

```

for $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = [ @array ];
}

```

Les crochets créent une référence à un nouveau tableau avec une *copie* de ce qui se trouve dans @array au moment de l'affectation. C'est ce que vous désiriez.

Notez que ceci produira quelque chose de similaire, mais c'est bien plus dur à lire :

```

for $i (1..10) {
    @array = 0 .. $i;
    @{$AoA[$i]} = @array;
}

```

Est-ce la même chose ? Eh bien, peut-être que oui, peut-être que non. La différence subtile est que lorsque vous affectez quelque chose entre crochets, vous êtes sûr que ce sera toujours une nouvelle référence contenant une nouvelle *copie* des données. Ça ne se passera pas forcément comme ça avec le déréférencement `@{$AoA[$i]}` du côté gauche de l'affectation. Tout dépend si `$AoA[$i]` était pour commencer indéfini, ou s'il contenait déjà une référence. Si vous aviez déjà peuplé @AoA avec des références, comme dans

```
$AoA[3] = \@another_array;
```

alors l'affectation avec l'indirection du côté gauche utiliserait la référence existante déjà présente :

```
@{$AoA[3]} = @array;
```

Bien sûr, ceci *aurait* l'effet "intéressant" de démolir @another\_list (Avez-vous déjà remarqué que lorsqu'un programmeur dit que quelque chose est "intéressant", au lieu de dire "intrigant", alors ça signifie que c'est "ennuyeux", "difficile", ou les deux ? :-).

Donc, souvenez-vous juste de toujours utiliser le constructeur de tableau ou de hachage `[]` ou `{}` et tout ira bien pour vous, même si ce n'est pas la solution optimale.

De façon surprenante, la construction suivante qui a l'air dangereuse marchera en fait très bien :

```

for $i (1..10) {
    my @array = somefunc($i);
    $AoA[$i] = \@array;
}

```

C'est parce que `my()` est plus une expression utilisée lors de l'exécution qu'une déclaration de compilation *en elle-même*. Cela signifie que la variable `my()` est recrée de zéro chaque fois que l'on traverse la boucle. Donc même si on a l'impression que vous stockez la même référence de variable à chaque fois, ce n'est pas le cas ! C'est une distinction subtile qui peut produire un code plus efficace au risque de tromper tous les programmeurs sauf les plus expérimentés. Je déconseille donc habituellement de l'enseigner aux débutants. En fait, sauf pour passer des arguments à des fonctions, j'aime rarement voir l'opérateur "donne-moi-une-référence" (backslash) utilisé dans du code. À la place, je conseille aux débutants (et à la plupart d'entre nous) d'essayer d'utiliser les constructeurs bien plus compréhensibles `[]` et `{}` au lieu de se reposer sur une astuce lexicale (ou dynamique) et un comptage de référence caché pour faire ce qu'il faut en coulisses. En résumé :

```

$AoA[$i] = [ @array ];      # habituellement mieux
$AoA[$i] = \@array;       # perilleux ; tableau declare
                           # avec my() ? Comment ?
@{ $AoA[$i] } = @array;    # bien trop astucieux pour la plupart
                           # des programmeurs

```

## 5 AVERTISSEMENT SUR LA PRÉCÉDENCE

En parlant de choses comme `@{ $AoA[$i] }`, les expressions suivantes sont en fait équivalentes :

```

$aref->[2][2]      # clair
$$aref[2][2]      # troublant

```

C'est dû aux règles de précedence de Perl qui rend les cinq préfixes déréférenciers (qui ont l'air de jurons : `$ @ * % &`) plus prioritaires que les accolades et les crochets d'indiciage postfixes ! Ceci sera sans doute un grand choc pour le programmeur C ou C++, qui est habitué à utiliser `*a[i]` pour indiquer ce qui est pointé par le *i-ème* élément de `a`. C'est-à-dire qu'ils prennent d'abord l'indice, et ensuite seulement déréférencent la structure à cet indice. C'est bien en C, mais nous ne parlons pas de C.

La construction apparemment équivalente en Perl, `$$aref[$i]` commence par déréférencer `$aref`, le faisant prendre `$aref` comme une référence à un tableau, et puis déréférence cela, et finalement vous donne la valeur du *i-ème* élément du tableau pointé par `$AoA`. Si vous vouliez la notion équivalente en C, vous devriez écrire `#{ $AoA[$i] }` pour forcer l'évaluation de `$AoA[$i]` avant le premier déréférencier `$`.

## 6 POURQUOI TOUJOURS UTILISER `use strict`

Ce n'est pas aussi effrayant que ça en a l'air, détendez-vous. Perl possède un certain nombre de caractéristiques qui vont vous aider à éviter les pièges les plus communs. La meilleure façon de ne pas être troublé est de commencer les programmes ainsi :

```

#!/usr/bin/perl -w
use strict;

```

De cette façon, vous serez forcé de déclarer toutes vos variables avec `my()` et aussi d'interdire tout "déréférencement symbolique" accidentel. Par conséquent, si vous faites ceci :

```

my $aref = [
    [ "fred", "barney", "pebbles", "bambam", "dino", ],
    [ "homer", "bart", "marge", "maggie", ],
    [ "george", "jane", "elroy", "judy", ],
];

print $aref[2][2];

```

Le compilateur marquera immédiatement ceci comme une erreur *lors de la compilation*, car vous accédez accidentellement à `@aref` qui est une variable non déclarée, et il vous proposerait d'écrire à la place :

```

print $aref->[2][2]

```

## 7 DÉBOGAGE

Avant la version 5.002, le débogueur standard de Perl ne faisait pas un très bon travail lorsqu'il devait imprimer des structures de données complexes. Avec la version 5.002 et les suivantes, le débogueur inclut plusieurs nouvelles caractéristiques, dont une édition de la ligne de commande ainsi que la commande `x` pour afficher les structures de données complexes. Par exemple, voici la sortie du débogueur avec l'affectation à `$AoA` ci-dessus :

```
DB<1> x $AoA
$AoA = ARRAY(0x13b5a0)
  0  ARRAY(0x1f0a24)
     0  'fred'
     1  'barney'
     2  'pebbles'
     3  'bambam'
     4  'dino'
  1  ARRAY(0x13b558)
     0  'homer'
     1  'bart'
     2  'marge'
     3  'maggie'
  2  ARRAY(0x13b540)
     0  'george'
     1  'jane'
     2  'elroy'
     3  'judy'
```

## 8 EXEMPLES DE CODE

Présentés avec peu de commentaires (ils auront leurs propres pages de manuel un jour), voici de courts exemples de code illustrant l'accès à divers types de structures de données.

## 9 TABLEAUX DE TABLEAUX

### 9.1 Déclaration d'un TABLEAU DE TABLEAUX

```
@AoA = (
    [ "fred", "barney" ],
    [ "george", "jane", "elroy" ],
    [ "homer", "marge", "bart" ],
);
```

### 9.2 Génération d'un TABLEAU DE TABLEAUX

```
# lecture dans un fichier
while ( <> ) {
    push @AoA, [ split ];
}

# appel d'une fonction
for $i ( 1 .. 10 ) {
    $AoA[$i] = [ somefunc($i) ];
}

# utilisation de variables temporaires
for $i ( 1 .. 10 ) {
    @tmp = somefunc($i);
    $AoA[$i] = [ @tmp ];
}

# ajout dans une rangée existante
push @{ $AoA[0] }, "wilma", "betty";
```

## 9.3 Accès et affichage d'un TABLEAU DE TABLEAUX

```
# un element
$AoA[0][0] = "Fred";

# un autre element
$AoA[1][1] =~ s/(\w)/\u$1/;

# affiche le tout avec des refs
for $aref ( @AoA ) {
    print "\t [ @$aref ],\n";
}

# affiche le tout avec des indices
for $i ( 0 .. $#AoA ) {
    print "\t [ @{$AoA[$i]} ],\n";
}

# affiche tous les elements un par un
for $i ( 0 .. $#AoA ) {
    for $j ( 0 .. ${ $AoA[$i] } ) {
        print "elt $i $j is $AoA[$i][$j]\n";
    }
}
```

## 10 HACHAGE DE TABLEAUX

### 10.1 Déclaration d'un HACHAGE DE TABLEAUX

```
%HoA = (
    flintstones    => [ "fred", "barney" ],
    jetsons        => [ "george", "jane", "elroy" ],
    simpsons       => [ "homer", "marge", "bart" ],
);
```

### 10.2 Génération d'un HACHAGE DE TABLEAUX

```
# lecture dans un fichier
# flintstones: fred barney wilma dino
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $HoA{$1} = [ split ];
}

# lecture dans un fichier avec plus de variables temporaires
# flintstones: fred barney wilma dino
while ( $line = <> ) {
    ($who, $rest) = split /\s*/, $line, 2;
    @fields = split ' ', $rest;
    $HoA{$who} = [ @fields ];
}

# appel d'une fonction qui retourne une liste
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoA{$group} = [ get_family($group) ];
}
```

```
# idem, mais en utilisant des variables temporaires
for $group ( "simpsons", "jetsons", "flintstones" ) {
    @members = get_family($group);
    $HoA{$group} = [ @members ];
}

# ajout de nouveaux membres a une famille existante
push @{ $HoA{"flintstones"} }, "wilma", "betty";
```

### 10.3 Accès et affichage d'un HACHAGE DE TABLEAUX

```
# un element
$HoA{flintstones}[0] = "Fred";

# un autre element
$HoA{simpsons}[1] =~ s/(\w)/\u$1/;

# affichage du tout
foreach $family ( keys %HoA ) {
    print "$family: @{ $HoA{$family} }\n"
}

# affichage du tout avec des indices
foreach $family ( keys %HoA ) {
    print "family: ";
    foreach $i ( 0 .. $#{$HoA{$family}} ) {
        print " $i = $HoA{$family}[$i]";
    }
    print "\n";
}

# affichage du tout trie par le nombre de membres
foreach $family ( sort { @{$HoA{$b}} <=> @{$HoA{$a}} } keys %HoA ) {
    print "$family: @{ $HoA{$family} }\n"
}

# affichage du tout trie par le nombre de membres et le nom
foreach $family ( sort {
    @{$HoA{$b}} <=> @{$HoA{$a}}
    ||
    $a cmp $b
} keys %HoA )
{
    print "$family: ", join(", ", sort @{ $HoA{$family} } ), "\n";
}
```

## 11 TABLEAUX DE HACHAGES

### 11.1 Déclaration d'un TABLEAU DE HACHAGES

```
@AoH = (
    {
        Lead    => "fred",
        Friend  => "barney",
    },
    {
        Lead    => "george",
        Wife    => "jane",
        Son     => "elroy",
    }
)
```

```

    },
    {
        Lead    => "homer",
        Wife    => "marge",
        Son     => "bart",
    }
);

```

## 11.2 Génération d'un TABLEAU DE HACHAGES

```

# lecture dans un fichier
# format : LEAD=fred FRIEND=barney
while ( <> ) {
    $rec = {};
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
    push @AoH, $rec;
}

# lecture dans un fichier sans variable temporaire
# format: LEAD=fred FRIEND=barney
while ( <> ) {
    push @AoH, { split /[\\s+=]/ };
}

# appel d'une fonction qui retourne une liste clé/valeur, comme
# "lead","fred","daughter","pebbles"
while ( %fields = getnextpairset() ) {
    push @AoH, { %fields };
}

# idem, mais sans variables temporaires
while (<>) {
    push @AoH, { parsepairs($_) };
}

# ajout d'un couple clé/valeur à un element
$AoH[0]{pet} = "dino";
$AoH[2]{pet} = "santa's little helper";

```

## 11.3 Accès et affichage d'un TABLEAU DE HACHAGES

```

# un element
$AoH[0]{lead} = "fred";

# un autre element
$AoH[1]{lead} =~ s/(\\w)/\\u$1/;

# affichage du tout avec des refs
for $href ( @AoH ) {
    print "{ ";
    for $role ( keys %$href ) {
        print "$role=$href->{$role} ";
    }
    print "}\\n";
}

```

```

# affichage du tout avec des indices
for $i ( 0 .. $#AoH ) {
    print "$i is { ";
    for $role ( keys %{ $AoH[$i] } ) {
        print "$role=$AoH[$i]{$role} ";
    }
    print ")\n";
}

# affichage du tout element par element
for $i ( 0 .. $#AoH ) {
    for $role ( keys %{ $AoH[$i] } ) {
        print "elt $i $role is $AoH[$i]{$role}\n";
    }
}

```

## 12 HACHAGES DE HACHAGES

### 12.1 Déclaration d'un HACHAGE DE HACHAGES

```

%HoH = (
    flintstones => {
        lead      => "fred",
        pal       => "barney",
    },
    jetsons      => {
        lead      => "george",
        wife      => "jane",
        "his boy" => "elroy",
    },
    simpsons     => {
        lead      => "homer",
        wife      => "marge",
        kid       => "bart",
    },
);

```

### 12.2 Génération d'un HACHAGE DE HACHAGES

```

# lecture dans un fichier
# flintstones: lead=fred pal=barney wife=wilma pet=dino
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $HoH{$who}{$key} = $value;
    }
}

# lecture dans un fichier, encore plus de variables temporaires
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    $rec = {};
    $HoH{$who} = $rec;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
}

```

```

# appel d'une fonction qui retourne un hachage clé/valeur
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoH{$group} = { get_family($group) };
}

# idem, mais en utilisant des variables temporaires
for $group ( "simpsons", "jetsons", "flintstones" ) {
    %members = get_family($group);
    $HoH{$group} = { %members };
}

# ajout de nouveaux membres a une famille existante
%new_folks = (
    wife => "wilma",
    pet  => "dino",
);

for $what (keys %new_folks) {
    $HoH{flintstones}{$what} = $new_folks{$what};
}

```

### 12.3 Accès et affichage d'un HACHAGE DE HACHAGES

```

# un element
$HoH{flintstones}{wife} = "wilma";

# un autre element
$HoH{simpsons}{lead} =~ s/(\w)/\u$1/;

# affichage du tout
foreach $family ( keys %HoH ) {
    print "$family: { ";
    for $role ( keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# affichage du tout un peu trie
foreach $family ( sort keys %HoH ) {
    print "$family: { ";
    for $role ( sort keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# affichage du tout trie par le nombre de membres
foreach $family ( sort { keys %{ $HoH{$b} } <=> keys %{ $HoH{$a} } } keys %HoH ) {
    print "$family: { ";
    for $role ( sort keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# etablissement d'un ordre de tri (rang) pour chaque role
$i = 0;
for ( qw(lead wife son daughter pal pet) ) { $rank{$_} = ++$i }

```

```
# maintenant affiche le tout trie par le nombre de membres
foreach $family ( sort { keys %{ $HoH{$b} } <=> keys %{ $HoH{$a} } } keys %HoH ) {
    print "$family: { ";
    # et affichage selon l'ordre de tri (rang)
    for $role ( sort { $rank{$a} <=> $rank{$b} } keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print ")\n";
}
}
```

## 13 ENREGISTREMENTS PLUS ÉLABORÉS

### 13.1 Déclaration d'ENREGISTREMENTS PLUS ÉLABORÉS

Voici un exemple montrant comment créer et utiliser un enregistrement dont les champs sont de types différents :

```
$rec = {
    TEXT      => $string,
    SEQUENCE => [ @old_values ],
    LOOKUP    => { %some_table },
    THATCODE => \&some_function,
    THISCODE => sub { $_[0] ** $_[1] },
    HANDLE    => \*STDOUT,
};

print $rec->{TEXT};

print $rec->{SEQUENCE}[0];
$last = pop @ { $rec->{SEQUENCE} };

print $rec->{LOOKUP}{"key"};
($first_k, $first_v) = each %{ $rec->{LOOKUP} };

$answer = $rec->{THATCODE}->($arg);
$answer = $rec->{THISCODE}->($arg1, $arg2);

# attention aux accolades de bloc supplémentaires sur la ref fh
print { $rec->{HANDLE} } "a string\n";

use FileHandle;
$rec->{HANDLE}->autoflush(1);
$rec->{HANDLE}->print(" a string\n");
```

### 13.2 Déclaration d'un HACHAGE D'ENREGISTREMENTS COMPLEXES

```
%TV = (
    flintstones => {
        series    => "flintstones",
        nights    => [ qw(monday thursday friday) ],
        members   => [
            { name => "fred",    role => "lead", age => 36, },
            { name => "wilma",   role => "wife", age => 31, },
            { name => "pebbles", role => "kid",  age => 4,  },
        ],
    },
),
```

```

jetsons    => {
    series  => "jetsons",
    nights  => [ qw(wednesday saturday) ],
    members => [
        { name => "george",  role => "lead", age  => 41, },
        { name => "jane",    role => "wife", age  => 39, },
        { name => "elroy",   role => "kid",  age  =>  9, },
    ],
},

simpsons   => {
    series  => "simpsons",
    nights  => [ qw(monday) ],
    members => [
        { name => "homer",  role => "lead", age  => 34, },
        { name => "marge",  role => "wife", age  => 37, },
        { name => "bart",   role => "kid",  age  => 11, },
    ],
},
);

```

### 13.3 Génération d'un HACHAGE D'ENREGISTREMENTS COMPLEXES

```

# lecture d'un fichier
# c'est plus facile a faire quand on dispose du fichier lui-meme
# dans le format de donnees brut explicite ci-dessus. perl analyse
# joyeusement les structures de donnees complexes si elles sont
# declarees comme des donnees, il est donc parfois plus facile de
# faire ceci

# voici un assemblage morceau par morceau
$rec = {};
$rec->{series} = "flintstones";
$rec->{nights} = [ find_days() ];

@members = ();
# on presume que ce fichier a la syntaxe champ=valeur
while (<>) {
    %fields = split /\[s=]+/;
    push @members, { %fields };
}
$rec->{members} = [ @members ];

# maintenant on se rappelle du tout
$TV{ $rec->{series} } = $rec;

#####
# maintenant, vous pourriez vouloir creer des champs
# supplementaires interessants incluant des pointeurs vers
# l'interieur de cette meme structure de donnees pour que
# si l'on en change un morceau, il se retrouve change partout
# ailleurs, comme par exemple un champ {kids} (enfants, NDT)
# qui serait une reference vers un tableau des enregistrements
# des enfants, sans avoir d'enregistrements dupliques et donc
# des soucis de mise a jour.
#####
foreach $family (keys %TV) {
    $rec = $TV{$family}; # pointeur temporaire
    @kids = ();
    for $person ( @{$rec->{members} } ) {

```

```

        if ($person->{role} =~ /kid|son|daughter/) {
            push @kids, $person;
        }
    }
    # SOUVENEZ-VOUS : $rec et $TV{$family} pointent sur les memes donnees !!
    $rec->{kids} = [ @kids ];
}

# vous avez copie le tableau, mais le tableau lui-meme contient des
# pointeurs vers des objets qui n'ont pas été copiés. Cela veut
# dire que si vous vieillissez bart en faisant

$TV{simpsons}{kids}[0]{age}++;

# alors ça changerait aussi
print $TV{simpsons}{members}[2]{age};

# car $TV{simpsons}{kids}[0] et $TV{simpsons}{members}[2]
# pointent tous deux vers la meme table de hachage anonyme sous-jacente

# imprime le tout
foreach $family ( keys %TV ) {
    print "the $family";
    print " is on during @{$TV{$family}{nights} }\n";
    print "its members are:\n";
    for $who ( @{$TV{$family}{members} } ) {
        print " $who->{name} ($who->{role}), age $who->{age}\n";
    }
    print "it turns out that $TV{$family}{lead} has ";
    print scalar ( @{$TV{$family}{kids} } ), " kids named ";
    print join (", ", map { $_->{name} } @{$TV{$family}{kids} } );
    print "\n";
}

```

## 14 Liens avec les bases de données

Il n'est pas possible de lier facilement une structure de données multiniveaux (tel qu'un hachage de hachages) à un fichier de base de données (fichier dbm). Le premier problème est que tous ces fichiers, à l'exception des GBM et des DB de Berkeley, ont des limitations de taille, mais il y a d'autres problèmes dûs la manière dont les références sont représentées sur le disque. Il existe un module expérimental, le module MLDBM, qui essaye de combler partiellement ce besoin. Pour obtenir le code source du module MLDBM allez voir sur le site CPAN le plus proche comme décrit dans le manuel *perlmodlib*.

## 15 VOIR AUSSI

le manuel *perlref*, le manuel *perllol*, le manuel *perldata* et le manuel *perlobj*.

## 16 AUTEUR

Tom Christiansen <[christ@perl.com](mailto:christ@perl.com)>

Dernière mise à jour: Wed Oct 23 04:57:50 MET DST 1996

## 17 TRADUCTION

### 17.1 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/>.

## 17.2 Traducteur

Roland Trique <[roland.trique@free.fr](mailto:roland.trique@free.fr)>

## 17.3 Relecture

Pascal Ethvignot <[pascal@encelade.frmug.org](mailto:pascal@encelade.frmug.org)>, Etienne Gauthier <[egauthie@capgemini.fr](mailto:egauthie@capgemini.fr)>, Paul Gaborit (Paul.Gaborit@enstimac.fr).

# 18 À propos de ce document

Ce document est la traduction française du document original 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.

*Ce document PDF est distribué selon les termes de la license Artistique de Perl. Toute autre distribution de ce fichier ou de ses dérivés impose qu'un arrangement soit fait avec le(s) propriétaire(s) des droits. Ces droits appartiennent aux auteurs du document original (lorsqu'ils sont identifiés dans la rubrique AUTEUR), aux traducteurs et relecteurs pour la version française et à moi-même pour la version PDF.*