—— Perl ——
— Cours 3/6 —

Entrées/sorties, références


Sylvain Lhullier
contact@formation-perl.fr
https://formation-perl.fr/


Plan des 4 premiers cours Perl


Opérateurs sur nom de fichiers

`-e`	existe	`-r`	lisible
`-f`	fichier normal	`-w`	écriture possible
`-d`	répertoire	`-x`	exécutable
`-l`	lien symbolique	`-o`	je suis le propriétaire

`-z`	fichier vide
`-s`	renvoie la taille en octets (**undef** si inexistant)

```
$file = '/usr/doc/vi/help';                    perldoc -f -X
my $taille = -s $file;
if( -f $file and -w $file ) { ... }
if( -f -w $file ) { ... } # chaînables depuis Perl 5.10
```

S'appliquent aussi sur les descripteurs de fichier (lire la suite)


Fonction `glob` : globalisation

`glob` renvoie une liste de noms de fichier selon les wildcards du shell.

```
my @t = glob('*.txt');
```

renvoie le nom de tous les fichiers `.txt` du répertoire courant

```
my @t = glob('*.pl *.pm');
my @t = glob('~/*.svg');

my @t = glob('/usr/share/doc/*/*.txt');
my @t = </usr/share/doc/*/*.txt>; # À éviter

foreach my $name ( glob('~/.* ~/*') ) {
   next if not -f $name;
   print "$name : ". ( -s $name ) ."\n";
}
```

Fin du programme

- La commande `exit` met fin au programme.
  `exit(2);` ⇒ code de retour du programme.
  Mais on s'en sert peu en Perl
- `die` affiche un message et met fin au programme.
  `die('Souci');` ⇒ affichage du nom de ficher et de la ligne,
  `die("Souci\n");` ⇒ affichage du seul message.
- `warn` affiche les mêmes messages et le programme continue.
- La variable magique `$!` vaut la dernière erreur système :
  - contexte numérique : le code errno,
  - contexte de chaîne / double-quote : le message localisé.

  Exemple typique : `die("$!");`

Fichiers : ouverture

`open(` *descripteur*, *mode*, *path* `)`

- le descripteur (*handle*) représente le fichier dans la suite du code (variable modifiée)
- Le mode d'ouverture peut être :

<	lecture		
>	écriture écrasement	>>	écriture ajout
+>	lecture/écrasement	+<	lecture/ajout

Exemples :
```
open( $fd, '<', 'perl.txt')
open( my $fd2, '>>', 'deamon.log')
open( my $fd3, '>', "/tmp/$name")
open( $h{$path}, '<', $path )
```


Fichiers : ouverture

La fonction `open` renvoie vrai ou faux :
```
if( !open($fd,'<','index.html') )
{ exit(1); }
```

Expression idiomatique en Perl :
```
open($fd,'<','index.html') or die("open: $!");
```
(évaluation paresseuse du *ou* logique)

V ou X vaut V
F ou ? vaut ?

Fichiers : ouverture, ancienne syntaxe

```
open(FILE,'>/tmp/filename.txt') or die("open: $!");
```

Descripteur : variable «*spéciale*» globale, non déclarée.
Nom libre usuellement en majuscules : `FILE`, `DATA`, `CONFIG`, etc.

`$fd` : variable locale à portée limitée `FILE` : variable globale
`$fd` : fermeture (`close`) à la destruction par le GC

Le mode d'ouverture et le nom du fichier sont concaténés.
Lecture par défaut. Dangereux : `open(FILE,$name)`
Injection si `$name = '>fichier'` (fichier de conf, base de données)

Fichiers : lecture

L'opérateur `<descripteur>` effectue une lecture ligne à ligne.

- En contexte de liste : **toutes** les lignes
  `@t = <$fd>;` À éviter, car peut être
  `foreach my $ligne (<$fd>) {...}` très gourmand en mémoire
- En contexte scalaire : une ligne
  - Lecture de la prochaine ligne : `$ligne = <$fd>;`
  - Lecture de **chaque** ligne :
    ```
    while( defined( my $ligne = <$fd> ) )
    { print $ligne; }
    ```
    Numéro de la ligne courante : `$.`

Le séparateur de ligne est défini dans la variable `$/`

Fonction `chomp`

- `chomp($x)`
  Supprime le dernier caractère de `$x` s'il s'agit d'une fin de ligne.
  (`$x` est modifiée). Peut prendre plusieurs arguments.
  !! ne pas écrire : `$x = chomp($x);`
  car `chomp` renvoie le nombre de caractères supprimés.

- ```
  open(my $fd,'<','fichier.txt') or die("open: $!");
  while( defined( my $ligne = <$fd> ) ) {
     chomp($ligne);
     ...
  }
  close($fd);
  ```


Fichiers ouverts au démarrage

- `STDIN` entrée standard
  `$v = <STDIN>;`

- `STDOUT` sortie standard
  `print` `printf` et `say` y écrivent par défaut

- `STDERR` sortie erreur standard
  `die` et `warn` y écrivent

Fichiers ouverts au démarrage : ARGV

`ARGV` simule le comportement standard des outils UNIX
On lit dans `<ARGV>` (ou `<>` opérateur diamant) les lignes :

- des fichiers donnés en argument au programme (open inutile)
- de l'entrée standard si pas d'argument ou si argument `-`

Exemple : `./prog.pl janvier.log fevrier.log`
`head -n 100 *.log | ./prog.pl`

```
while( defined( my $ligne = <> ) ) { print $ligne; }
```
⇒ pratique pour faire un filtre !

Nom du fichier en cours de lecture : `$ARGV`
Numéro de la ligne (dans le flux global) : `$.`
```

Fichiers

Écriture :

- `print $fd "toto\n";` (pas de virgule après le descripteur)
  Renvoie faux en cas d'erreur, vrai sinon
- `printf $fd '%03d', $i;`
- `say $fd 'toto';`

Fermeture : `close( $fd ); # Sinon appel par le GC`

Caractère / binaire :

- `$c = getc($fd);` Lecture d'un octet
- `$len = read($fd,$buffer,512);` Lecture d'un buffer
- `write` existe, mais n'est pas l'inverse de `read`
  ⇒ voir les fonctions `sysopen sysread syswrite close`

Fichiers : positionnement

`seek` *filehandle, position, whence*
Déplacement dans un fichier :
→ position en nombre de caractères
→ en relatif selon la valeur de *whence* :

- `SEEK_SET` : depuis le début du fichier
- `SEEK_CUR` : depuis la position actuelle
- `SEEK_END` : depuis la fin du fichier

```
use Fcntl; # En haut du script
seek($fd, 0, SEEK_SET);
```


Exemple de manipulation de fichiers

```
open(my $fd,'<',"$filename.txt") or die("open: $!");
my %total = ();
while( defined( my $ligne = <$fd> ) ) {
   my @mots = split( /\W+/, $ligne );
   foreach my $mot (@mots) {
      $total{lc $mot}++;
   }
}
close($fd);
foreach my $mot (sort keys %total) {
   print "$mot est inclus $total{$mot} fois.\n";
}
```


`binmode` : mettre un flux en mode binaire

Fonctions `print/printf` et opérateur `<$fd>` : mode texte
⇒ transformation des retours-chariot selon le système d'exploitation.
⇒ pratique pour la manipulation du texte, moins pour le binaire.

La fonction `binmode` met un flux en mode binaire.

```
open(my $fd,'>','image.jpg') or die("open: $!");
binmode $fd;
print $fd $contenu_image;
close $fd;
```

Sinon, les données binaires de l'image peuvent être corrompues.
NB: `chomp` ne fonctionne pas bien dans ce type de conditions.

Fichiers et UTF-8

Pour le traitement de fichiers en UTF-8, il faut ouvrir ainsi :

```
open(my $fd, '<:encoding(UTF-8)',      'fichier-utf8.txt')
open(my $fd, '<:encoding(iso-8859-1)', 'fichier-latin1.txt')
```

Pour que tout appel à open() utilise automatiquement un encodage :

```
use open ':encoding(UTF-8)'; # En haut du script
```

Pour un fichier déjà ouvert, notamment la sortie standard :

```
binmode $fd,     ':encoding(UTF-8)';
binmode STDOUT, ':encoding(UTF-8)';
```


Fichiers interface objet `IO::File`

```
use IO::File;
my $fd = IO::File->new( $path, '<')
         or die "open $path: $!";
while( defined( my $ligne = $fd->getline ) ) {
   print "$ligne\n";
}
# $fd->print("texte\n");
$fd->close;
```

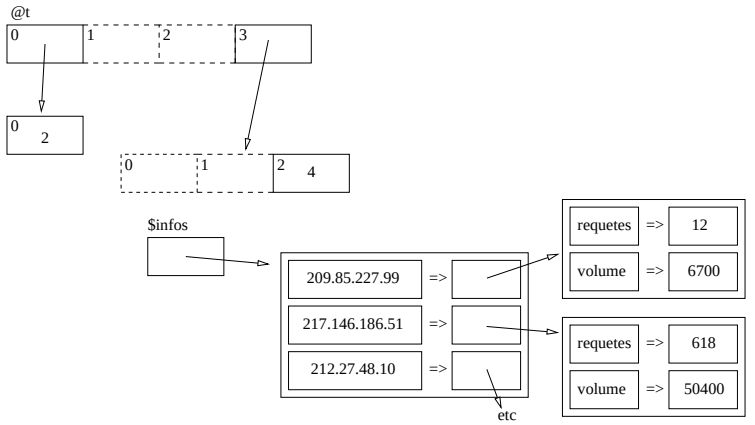
`IO::File` hérite de `IO::Handle` et de `IO::Seekable`

Fichiers interface objet `IO::Handle`

```
$io = IO::Handle->new();
$io = IO::Handle->new_from_fd ( FD, MODE );
$io->close                          $io->eof
$io->fileno                         $io->getc
$io->fcntl( FUNCTION, SCALAR )      $io->stat
$io->format_write( [FORMAT_NAME] )
$io->ioctl( FUNCTION, SCALAR )
$io->read( BUF, LEN, [OFFSET] )
$io->print( ARGS )
$io->printf( FMT, [ARGS] )
$io->say( ARGS )           $io->truncate( LEN )
$io->sysread( BUF, LEN, [OFFSET] )
$io->syswrite( BUF, [LEN, [OFFSET]] )
```


Références

Motivation : construire de puissantes structures de données



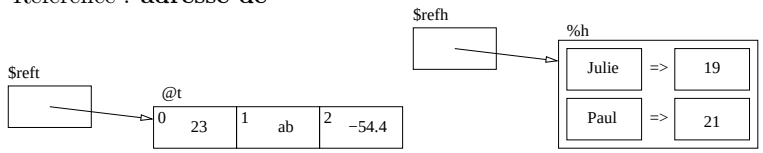
Références : plan du cours

Étapes :

- Syntaxe (passage obligé)
- Références anonymes (puissance et concision)
  ⇒ allocation dynamique
- Introspection avec `ref` (parfois utile)
- Affichage avec `Data::Dumper` (très pratique)
- Autovivification via références (puissance++ et concision++)

Références : principes

Référence : **adresse de**



Gestion mémoire sûre, pas d'arithmétique sur références, etc

Référence = valeur **scalaire** ⟹ stockage dans une variable scalaire

Java : référence C : pointeur C++ : référence/pointeur

Références : prise de référence

Déclaration et affectation :

```
my $refscalaire = \$toto;
my $reftableau  = \@tableau;
my $refhash     = \%hash;
my $reffonction = \&fonction;
my $reffichier  = \*STDOUT;  # open($fd,...) $fd=référence
```

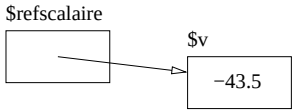
Affichage :

```
print "$refscalaire\n";   # SCALAR(0x80ff4f0)
print "$reftableau\n";    # ARRAY(0x80fcaf8)
print "$refhash\n";       # HASH(0x80fcb28)
print "$reffonction\n";   # CODE(0x80ff514)
print "$reffichier\n";    # GLOB(0x80fca14)
```


Utilisation des références sur scalaire

```
my $v = -43.5;
my $refscalaire = \$v;
print "$v\n";
print "$refscalaire\n";
print "$$refscalaire\n";
$$refscalaire = 'ok';
print "$v\n";
```

```
sub f {
   my ($r) = @_;
   $$r = 'hello';
}
f($refscalaire);
f(\$v);
```

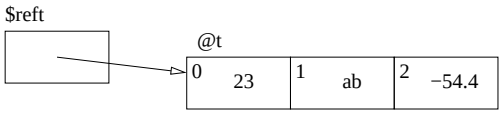


⟹ fonction swap

Référence vers une constante : `\23.4`

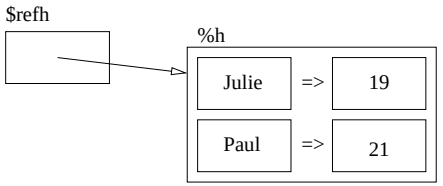
Utilisation des références sur tableau

```
my @t = (23, 'ab', -54.4);
my $reft = \@t;
foreach my $e (@$reft) { ... }
my @t2 = @$reft;
@$reft = ('z', 2);
$$reft[1] = 9;
$reft->[1] = 6;
```



Utilisation des références sur hash

```
my %h = ( Paul => 21, Julie => 19 );
my $refh = \%h;
my @clefs = keys %$refh;
my %h2 = %$refh;
$$refh{'Luc'} = '10';
$refh->{'Jacques'} = 33;
foreach my $k (keys %$refh) # $k est une chaîne de caractères
{  print "$k $refh->{$k}\n"; }
```



Utilisation des références avec les fonctions

```
sub traitement {
   my ($rt,$rh) = @_;
   foreach my $e (@$rt) { print "$e\n"; }
   $rt->[0] = -23;
   foreach my $k (keys %$rh) { print "$k $rh->{$k}\n"; }
   $rh->{Jacques} = 33;
}
my @t = (23, 'ab', -54.4);
my %h = ( Paul => 21, Julie => 19 );
traitement( \@t, \%h );
my $reft = \@t;
my $refh = \%h;
traitement( $reft, $refh );
```


Références : premier récapitulatif des syntaxes

Structure de données	Prise de référence	Déréférencement global	Déréférencement élémentaire
Scalaire	`$refs=\$s`	`$$refs`	
Tableau	`$reft=\@t`	`@$reft`	`$reft->[$i]`
Hash	`$refh=\%h`	`%$refh`	`$refh->{$k}`

Utilisation des références sur fonction

```
sub f {
   my ($p) = @_;
   print "Ok $p\n";
}
my $r = \&f;
&$r('Paul');
$r->('Paul');
```


Utilisation des références sur fichier

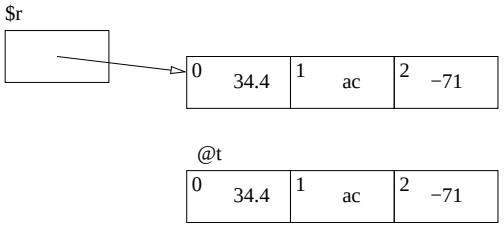
```
open(my $fd,'>','toto') or die($!);
# $fd est une référence

my $r = \*STDOUT;
print $r "ok\n";
sub f {
   my ($f) = @_;
   print $f "ok\n";    # En lecture  <$f>
}
f($r);      f(\*STDOUT);
f($fd);
close($r);
```


Références anonymes sur tableau

Pas d'existence dans l'espace de nom.

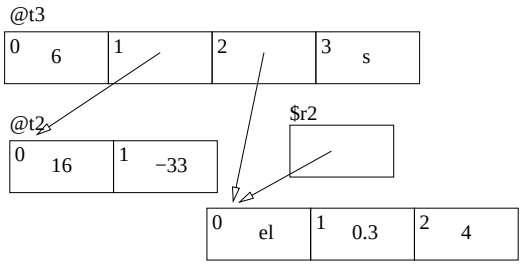
- `$r = [34.4, 'ac', -71];`
  $\neq$ `[]` opérateur d'indice sur tableau.



Rappel : `@t = (34.4, 'ac', -71);`

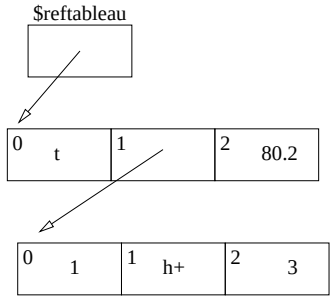
Références anonymes sur tableau

- `my @t2 = (16,-33);`
  `my $r2 = ['el',0.3,4];`
  `my @t3 = (6, \@t2, $r2, 's');`



Références anonymes sur tableau

- `$reftableau = ['t', [1,'h+',3], 80.2];`



Rappel : `@t = ('t', (1,'h+',3), 80.2); # Aplatissement`

Références anonymes sur tableau (suite)

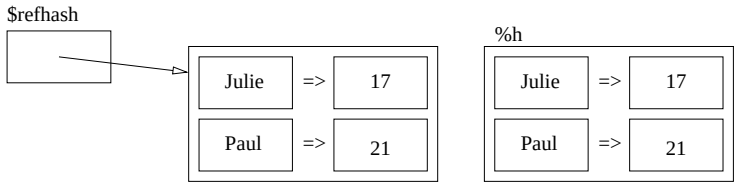
- Accéder à un élément :
  `$reftableau->[0]` vaut `'t'`.
  `$reftableau->[1]->[2]` vaut 3.

- Parcourir les tableaux :

```
foreach my $v ( @$reftableau ) {
   print "$v\n";
}
foreach my $v ( @{$reftableau->[1]} ) {
   print "$v\n";
}
```


Références anonymes sur hash

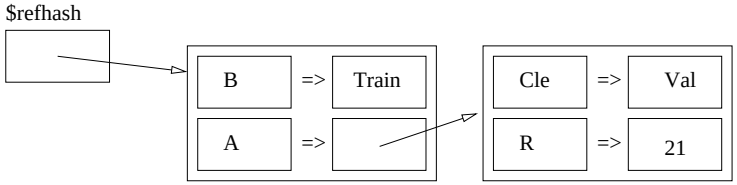
-
```
$refhash = { Paul  => 21,   # Usage de la double-flèche
             Julie => 17 }; # Attention au point-virgule !
```
$\neq$ `{}` opérateur de clef sur hash.



Rappel : `%h = (Paul=>12,Julie=>17);`

Références anonymes sur hash (suite)

-
```
$refhash = {
              A => { Cle=>'Val', R=>21 },
              B => 'Train'
           };
```



Références anonymes sur hash (suite)

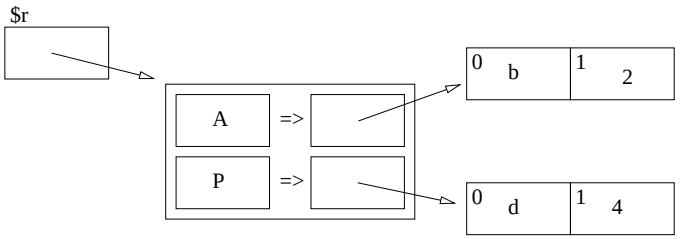
- Accéder à un élément :
  `$refhash->{B}` vaut 'Train'.
  `$refhash->{A}->{R}` vaut 21.

- Parcourir les tables de hash :

```
foreach my $k (keys %$refhash) {
   print "$k: $refhash->{$k}\n";
}
foreach my $k (keys %{$refhash->{A}}) {
   print "$k: $refhash->{A}->{$k}\n";
}
```


Références anonymes (suite)

-
```
my $r = {
           P => [ 'd', 4 ],
           A => [ 'b', 2 ],
         };
```



`$r->{A}->[1]` vaut 2.

Références anonymes sur tableau ou hash vide

- Référence vers un tableau vide :
  `$reftableau = [];`

- Référence vers une table de hachage vide :
  `$refhash = {};`

Références anonymes sur fonction

-
```
my $reffonction = sub { print "Ok\n"; };
```

-
```
my $reffonction2 = sub {
   my ($r,$v) = @_;
   print "($r,$v)\n";
   return $r+$v;
};
```

-
```
$x = $reffonction2->(3,56);
```

À quoi servent les références anonymes ?

Gestion **dynamique** de la mémoire (malloc en C).
Ne pas dépendre de la visibilité des variables des fonctions.

```
sub tache {
   my ($p1,$p2) = @_;
   my $rh = {};
   my $rt = [];
   ... remplissage de $rh et $rt ...
   return ($rh,$rt);
}
my ($refHash,$refTableau) = tache(1,'a');
print $refHash->{clef}, $refTableau->[0];
```


Références anonymes

```
sub f1 {
   my @t = (43,'ez');
   return \@t;
}
my $ref1 = f1();
print "$ref1->[1]\n";
```

```
sub f2 {
   my %h = (J=>2,P=>7);
   return \%h;
}
my $ref2 = f2();
print "$ref2->{J}\n";
```

Renvoyer une référence vers une variable locale : autorisé en Perl.

Suppression lors qu'aucune référence n'existe.
$\Longrightarrow$ Garbage collector

Références : récapitulatif complet des syntaxes

Structure de données	Prise de référence	Création anonyme	Déréférencement global	Déréférencement élémentaire
Scalaire	`$refs=\$s`		`$$refs ${$rsC}`	
Tableau	`$reft=\@t`	`$reft=[...]`	`@$reft @{$rtC}`	`$reft->[$i]`
Hash	`$refh=\%h`	`$refh={...}`	`%$refh %{$rhC}`	`$refh->{$k}`
Fonction	`$reff=\&f`	`$reff=sub{}`		`$reff->($v)`

NB: Le sigil indique le type de l'expression.

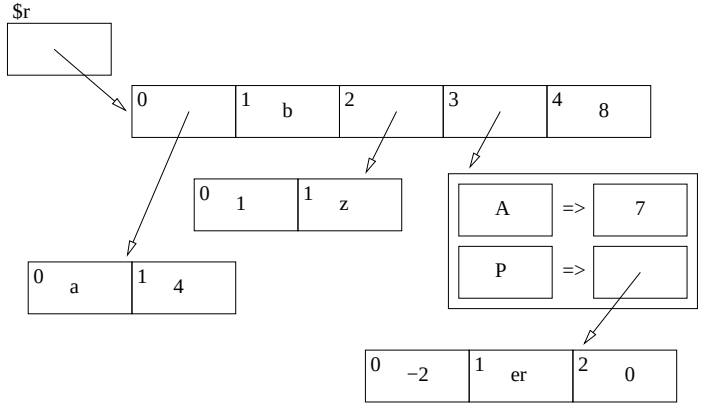
Références : opérateur `ref`

Cet opérateur indique le type de référence de son paramètre.
Selon la valeur `ref($r)` on sait que `$r` est :

- (faux) n'est pas une référence (scalaire "*normal*"),
- `'SCALAR'` une référence vers un scalaire,
- `'ARRAY'` une référence vers un tableau,
- `'HASH'` une référence vers une table de hachage,
- `'CODE'` une référence vers une fonction,
- `'GLOB'` une référence vers un fichier.

Références : opérateur ref

```
my $r = [ ['a',4], 'b', [1,'z'], {P=>[-2,'er',0],A=>7}, 8 ];
```



Références : opérateur ref

```
my $r = [ ['a',4], 'b', [1,'z'], {P=>[-2,'er',0],A=>7}, 8 ];
foreach my $p (@$r) {
   if( ref($p) eq 'ARRAY' ) {
      print '( ';
      foreach my $v (@$p)
      { print "$v "; }
      print ")\n";
   } elsif( ref($p) eq 'HASH' ) {
      foreach my $k (keys %$p)
      { print "$k : $p->{$k}\n"; }
   } elsif( !ref($p) ) {
      print "$p\n";
   }
}
```

```
( a 4 )
b
( 1 z )
P : ARRAY(0x8100c20)
A : 7
8
```


Références : Affichage en profondeur

Affichage complet d'une structure :

```
use Data::Dumper; # En haut de fichier

print Dumper($r);
print Dumper(\%h);
```

`Dumper` renvoie une chaîne de caractères représentant les données présentes au bout de la référence.

- Cette chaîne est directement intégrable dans un code Perl.
- Gestion des circularités, des éléments référencés plusieurs fois, etc.
- Options d'indentation ...

`perldoc Data::Dumper`

Références : type complexe

```
$r = { Paul => { Tel => '01.23.13.54',
                 Adr => '14, rue Pasteur',
                 Enfants => [ 'Julien', 'Laura', ], },
       Anne => { Tel => '02.74.10.40',
                 Adr => '5, bd Victor Hugo',
                 Enfants => [ 'Marine', 'Anne', 'Luc', ], },
       Robert=>{ Tel => '06.94.28.88',
                 Adr => '6, rue du Parc',
                 Enfants => [ ], },
     };
```

`$r->{Paul}->{Enfants}->[0]` Prénom du 1er enfant de Paul

`$r->{Paul}{Enfants}[0]` idem car `->` optionnelle entre `{} []`

Q: Comment afficher tous les enfants de Paul ?

Notez les dernières virgules.

Autovivification de structure via référence

Rappel d'autovivification : `my %h = ();`

```
foreach my $m (@mots) { $h{$m}++; }
  # if(!exists $h{$m}) {$h{$m}=1} else {$h{$m}++}
```

L'autovivification fonctionne via référence pour la structure :

```
my $r1;                 my $r2;
$r1->{$mot}=4;          $r2->[0]=4;
$r1->{$mot}+=4;         $r2->[0]+=4;
$r1->{$mot}++;          $r2->[0]++;
```

La première fois : création de la structure anonyme réclamée.
Ensuite, réutilisation de la structure qui existe :

```
$r1->[0] erreur!        $r2->{$k} erreur!

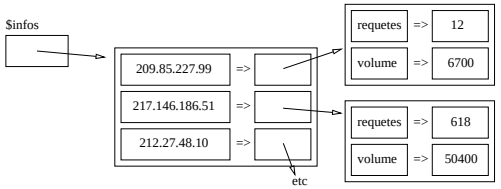
my $r;  $r->[0]{data}[3]{foo} = 'a';
```


Autovivification via références : Consolidation de données

Journaux Apache : nb requêtes et volume transféré pour chaque hôte

```
my $infos = {};
while(......) { # Lecture et analyse de chaque ligne
   $infos->{$ip}{requetes}++; # Création de la hash si besoin
   $infos->{$ip}{volume} += $nbOctets;
}
```

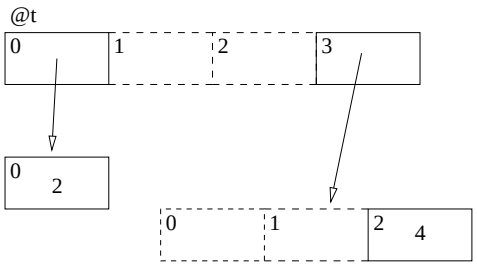
Pas la peine de tester la présence de la structure avant d'ajouter.



Autovivification via références : Tableau à n dimensions

```
my @t;
$t[0][0] = 2;          $t[0]->[0]
$t[3][2] = 4;
```

Semble fonctionner comme par magie :-)



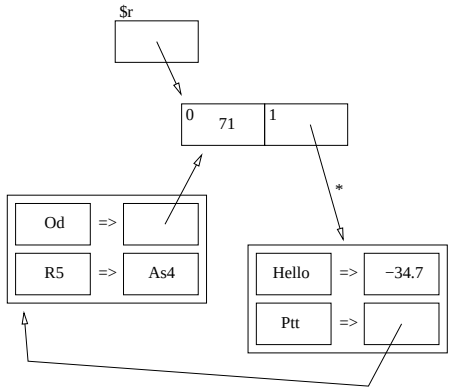
Références circulaires

```
my $r = [
  71,
  {
    Hello=>-34.7,
    Ptt  => {R5=>'As4'}
  }
];
$r->[1]{Ptt}{Od} = $r;
```

Pour libérer cette structure :

```
$r->[1] = undef;
$r = undef;
```

⇒ Attention aux fuites mémoire



Tranche via référence

```
my $rt = [ 'a', 'b', 'c', ];
print join(',', @{$rt}[2,0] )."\n";

my $rh = {
    X=>1,
    Y=>2,
    Z=>3,
};
print join(',', @{$rh}{'Z','X'} )."\n";
```


Tri : Transformée schwartzienne

Méthode de cache pour le tri popularisée par Randal L. Schwartz.
But : accélérer un tri effectué selon un calcul qui coûte cher.
Idée : effectuer une seule fois le calcul pour chaque élément.

```
@fruits = ('pomme', 'groseille', 'melon', 'kiwi');
@fruits = sort { length($a) <=> length($b) } @fruits;
@fruits = map { $_->[0] }
            sort { $a->[1] <=> $b->[1] }
              map { [ $_, length($_) ] }
                @fruits;
```

Limitation : le calcul à cacher doit se faire sur un seul élément.
Question : comment affiner le tri par l'ordre alphabétique des fruits ?