Perl

5

Olivier Thauvin

18 septembre 2008

1 Presentation

2 Variables

3 Syntaxe

4 Chaines de caractères

5 Variables bis

6 Fin

Plan

1. Presentation
2. Variables
3. Syntaxe
4. Chaines de caractères
5. Variables bis
6. Fin

PERL

Perl, écrit par Larry Wall en 1987.

PERL

Perl, écrit par Larry Wall en 1987.
Perl 5.10 maintenu par Rafael Garcia-Suarez

PERL

Perl, écrit par Larry Wall en 1987.
Perl 5.10 maintenu par Rafael Garcia-Suarez
Dernière version : 5.10.0

PERL

Perl, écrit par Larry Wall en 1987.
Perl 5.10 maintenu par Rafael Garcia-Suarez
Dernière version : 5.10.0

PERL ne signifie rien

Practical Extraction and Report Language (*langage pratique d'extraction et de génération de rapports*)
ou
Pathetically Eclectic Rubbish Lister (*collectionneur de déchets pathétiquement éclectiques*)

PERL

Perl, écrit par Larry Wall en 1987.
Perl 5.10 maintenu par Rafael Garcia-Suarez
Dernière version : 5.10.0

PERL ne signifie rien

Practical Extraction and Report Language (*langage pratique d'extraction et de génération de rapports*)
ou
Pathetically Eclectic Rubbish Lister (*collectionneur de déchets pathétiquement éclectiques*)

Perl 5 !

Perl 6 sera très différent

Pourquoi PERL ?

- langage scripté, non compilé

Pourquoi PERL ?

- langage scripté, non compilé
- gestion de la mémoire dynamique et automatique

Pourquoi PERL ?

- langage scripté, non compilé
- gestion de la mémoire dynamique et automatique
- portable (existe sous Unix*, Windows, OS/2, VMS. . .)

Pourquoi PERL ?

- langage scripté, non compilé
- gestion de la mémoire dynamique et automatique
- portable (existe sous Unix*, Windows, OS/2, VMS. . .)
- est orienté objet

Pourquoi PERL ?

- langage scripté, non compilé
- gestion de la mémoire dynamique et automatique
- portable (existe sous Unix*, Windows, OS/2, VMS. . .)
- est orienté objet
- possède un grand nombre de modules additionnels

Pourquoi PERL ?

- langage scripté, non compilé
- gestion de la mémoire dynamique et automatique
- portable (existe sous Unix*, Windows, OS/2, VMS. . .)
- est orienté objet
- possède un grand nombre de modules additionnels
- est très documenté

Pourquoi PERL ?

- langage scripté, non compilé
- gestion de la mémoire dynamique et automatique
- portable (existe sous Unix*, Windows, OS/2, VMS. . .)
- est orienté objet
- possède un grand nombre de modules additionnels
- est très documenté
- la communauté perl est conviviale

Pourquoi PERL ?

- langage scripté, non compilé
- gestion de la mémoire dynamique et automatique
- portable (existe sous Unix*, Windows, OS/2, VMS. . .)
- est orienté objet
- possède un grand nombre de modules additionnels
- est très documenté
- la communauté perl est conviviale

Utilisation :

- petits scripts type shell

Pourquoi PERL ?

- langage scripté, non compilé
- gestion de la mémoire dynamique et automatique
- portable (existe sous Unix*, Windows, OS/2, VMS. . .)
- est orienté objet
- possède un grand nombre de modules additionnels
- est très documenté
- la communauté perl est conviviale

Utilisation :

- petits scripts type shell
- des programmes complexes

Pourquoi PERL ?

- langage scripté, non compilé
- gestion de la mémoire dynamique et automatique
- portable (existe sous Unix*, Windows, OS/2, VMS. . .)
- est orienté objet
- possède un grand nombre de modules additionnels
- est très documenté
- la communauté perl est conviviale

Utilisation :

- petits scripts type shell
- des programmes complexes
- des services de toutes sortes

Pourquoi PERL ?

- langage scripté, non compilé
- gestion de la mémoire dynamique et automatique
- portable (existe sous Unix*, Windows, OS/2, VMS. . .)
- est orienté objet
- possède un grand nombre de modules additionnels
- est très documenté
- la communauté perl est conviviale

Utilisation :

- petits scripts type shell
- des programmes complexes
- des services de toutes sortes
- des sites web
- . . .

Exemples

Exemple de script :

```
#!env perl

print "Hello World\n";
```

Exemples

Exemple de script :

```
#!env perl

print "Hello World\n";
```

One-liner

```
perl -e 'print "Hello World\n"'
```

Obfusacte perl

Sinon ça peut aussi ressembler à ça :

bestever.pl de Philippe Bruhat (Book)

```
/;{}def/#{def}def/$_={/Times-Bold exch selectfont}#/_{rmoveto}#/"{dup}#/*/!/$
;/q{exch}#/x ; {/J q #}#/.{/T q #}#{stringwidth}#{}#{}# 14 string dup dup dup
260 40 moveto 90 rotate ; %/}};$0='"\e[7m \e[0m"';@ARGV=split//,reverse
q(ThePerl). q(Journal) x 220 ; q ; 0 T putinterval exch 7 J putinterval ;
 ; $_= q /m$ pop T($*!$"=!$ " )pop " * true% ? $ " $!" "  !!  !! % !" !"    !
! charpath {!"""}pop $ pop{""!}pop ! neg{!#}pop 220 ! neg _{!!}pop J false %T
charpath  clip " pop 0 " moveto 6{!!}pop $_= 105{!!}pop {$ ! $ " !  #! ##}
pop{dup dup $ ! " pop pop q{"}pop 22{dup show}repeat {"}pop q 22 mul{$ "} pop
neg{!#! $ "}pop ! 8 .65 mul{$ # # $}pop ! neg{"}pop  _ pop{"}pop } repeat pop
" {  $ " ! ! ! $ " ! !" "#"  #"!"""""! #" " # "m/;@ARGV=(@ARGV[-14..-1])x50;q}
 0 "%};s/m[ou]|[-\dA-ln-z.\n_{}]|\$_=//gx;s/(.)(?{$*=''})/('$*.=''.(++$#
%2?'':"$0;").'pop;')x(ord($1)-31).'$*'/gee;s/((.(\e\[.m)*|.){77})/$1\n/g;print
; sub showpage {}
```

Avant-propos

Ce dont je ne vais pas parler. . .

Avant-propos

Ce dont je ne vais pas parler. . .

- Plain Old Documentation - documenter son code

Avant-propos

Ce dont je ne vais pas parler. . .

- Plain Old Documentation - documenter son code
- Espaces de noms

Avant-propos

Ce dont je ne vais pas parler. . .

- Plain Old Documentation - documenter son code
- Espaces de noms
- Ecriture de modules

Avant-propos

Ce dont je ne vais pas parler. . .

- Plain Old Documentation - documenter son code
- Espaces de noms
- Ecriture de modules
- Objets

Avant-propos

Ce dont je ne vais pas parler. . .

- Plain Old Documentation - documenter son code
- Espaces de noms
- Ecriture de modules
- Objets
- XS : pont du C <=> perl

Avant-propos

Ce dont je ne vais pas parler. . .

- Plain Old Documentation - documenter son code
- Espaces de noms
- Ecriture de modules
- Objets
- XS : pont du C <=> perl
- tests sur les modules et objets

Avant-propos

Ce dont je ne vais pas parler. . .

- Plain Old Documentation - documenter son code
- Espaces de noms
- Ecriture de modules
- Objets
- XS : pont du C <=> perl
- tests sur les modules et objets
- de la tonne de modules super géniale qui fait choses épatantes

Avant-propos

Ce dont je ne vais pas parler. . .

- Plain Old Documentation - documenter son code
- Espaces de noms
- Ecriture de modules
- Objets
- XS : pont du C <=> perl
- tests sur les modules et objets
- de la tonne de modules super géniale qui fait choses épatantes
- obfuscate perl

Avant-propos

Ce dont je ne vais pas parler. . .

- Plain Old Documentation - documenter son code
- Espaces de noms
- Ecriture de modules
- Objets
- XS : pont du C <=> perl
- tests sur les modules et objets
- de la tonne de modules super géniale qui fait choses épatantes
- obfuscate perl
- des trucs super gores qu'on peut faire en perl, bien pire que l'obfuscate

Je sais, vous êtes déçus !

There's More Than One Way To Do It !

Larry Wall

Plan

1. Presentation
2. Variables
   - Variables de Perl
   - Scalaire
   - Les tableaux
   - Les tables de hashage
   - Les listes
   - les variables magiques
3. Syntaxe
4. Chaines de caractères
5. Variables bis

2 Variables
- Variables de Perl
- Scalaire
- Les tableaux
- Les tables de hashage
- Les listes
- les variables magiques

types de variables

15 types de variables :

types de variables

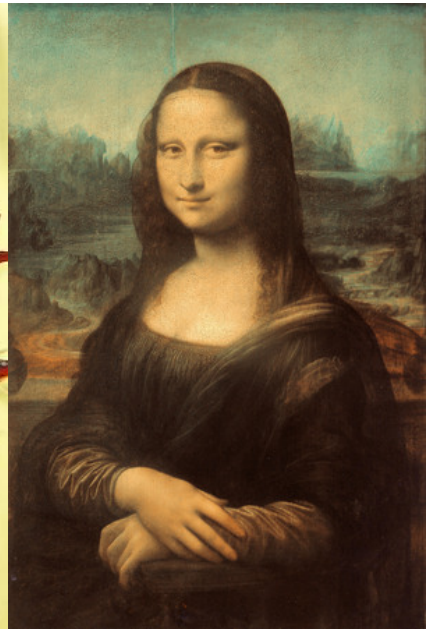
15 types de variables :

- scalaires (scalar)

types de variables

15 types de variables :

- scalaires (scalar)
- tableaux (array)



types de variables

15 types de variables :

- scalaires (scalar)
- tableaux (array)
- hash (hash)

types de variables

15 types de variables :

- scalaires (scalar)
- tableaux (array)
- hash (hash)

Et les autres on s'en moque...

2 Variables
- Variables de Perl
- Scalaire
- Les tableaux
- Les tables de hashage
- Les listes
- les variables magiques

scalaire

Un **scalaire** est une simple valeur :

scalaire

Un **scalaire** est une simple valeur :

- *undef*

scalaire

Un **scalaire** est une simple valeur :

- *undef*
- définie :
  - un chiffre, nombre
  - une chaine de caractères
  - ...

scalaire

Un **scalaire** est une simple valeur :

- *undef*
- définie :
  - un chiffre, nombre
  - une chaine de caractères
  - ...

Convertion

Perl évalue et converti les valeurs numériques au vol !

scalaire

Un **scalaire** est une simple valeur :

- *undef*
- définie :
  - un chiffre, nombre
  - une chaine de caractères
  - ...

Convertion

Perl évalue et converti les valeurs numériques au vol !

Exemple de valeur de scalaire

```
$rien = undef;
$chiffre = 42; $chaine = "1";
print $chaine + $chiffre;
print $chaine . $chiffre;
```

Scalair : Vrai ou Faux ?

Valeur *fausses* : *undef*, 0, "0", ""
Valeur *vraies* : le reste : 1, "1", "un chien ?"

Scalair : Vrai ou Faux ?

Valeur *fausses* : *undef*, 0, "0", ""
Valeur *vraies* : le reste : 1, "1", "un chien ?"

Exemple de test

```
$valeur = "0";
if ($valeur) # faux, "0" => 0
if ($valeur == 0) # vrai "0" => 0
if ($valeur eq "") # faux, chaine non vide
```

Scalair : Vrai ou Faux ?

Valeur *fausses* : *undef*, 0, "0", ""
Valeur *vraies* : le reste : 1, "1", "un chien ?"

Exemple de test

```
$valeur = "0";
if ($valeur) # faux, "0" => 0
if ($valeur == 0) # vrai "0" => 0
if ($valeur eq "") # faux, chaine non vide
```

0 but True

```
$valeur = "0E0";
```

2 Variables
- Variables de Perl
- Scalaire
- Les tableaux
- Les tables de hashage
- Les listes
- les variables magiques

Tableau (array)

Les tableaux sont des piles de scalaires

```
$valeur = 3;
@tableau = ($valeur, "chaines");
```

Tableau (array)

Les tableaux sont des piles de scalaires

```
$valeur = 3;
@tableau = ($valeur, "chaines");
```

Ils sont dynamiques, et a une seule dimension

Tableau : Exemples

Utilisation :

```
@tableau = (1, 2, 3);
$tableau[3] = 4;
$tableau[-1]; # dernier élément
```

Tableau : Exemples

Utilisation :

```
@tableau = (1, 2, 3);
$tableau[3] = 4;
$tableau[-1]; # dernier élément
```

Attention au $

```
@tableau;
$tableau[1];
```

Tableau : Exemples

Utilisation :

```
@tableau = (1, 2, 3);
$tableau[3] = 4;
$tableau[-1]; # dernier élément
```

Attention au $

```
@tableau;
$tableau[1];
```

Nombre d'éléments : `scalar(@tableau)`
Dernier indice : `$#tableau`

Tableau : Exemples

Utilisation :

```
@tableau = (1, 2, 3);
$tableau[3] = 4;
$tableau[-1]; # dernier élément
```

Attention au $

```
@tableau;
$tableau[1];
```

Nombre d'éléments : `scalar(@tableau)`
Dernier indice : `$#tableau`

Les bouts du tableau

shift <=	`0$$$$$n`	<= push
unshift =>	`0$$$$$n`	=> pop

2 Variables
- Variables de Perl
- Scalaire
- Les tableaux
- Les tables de hashage
- Les listes
- les variables magiques

hash

tables de hashage

Ce sont des tableaux indexés `clef => valeur`.
Ils sont dynamiques.

hash

tables de hashage

Ce sont des tableaux indexés `clef => valeur`.
Ils sont dynamiques.

- la clef est une valeur textuelle simple
- la valeur est un **scalaire**
- l'ordre des clefs n'est pas conservé ! ! !

hash : exemple

Utiliser tableau de hashage :

```
%hash = ( foo => "un",
          bar => 2, );
print $hash{'bar'};
$clef = 'clef'; $hash{$clef};
```

hash : exemple

Utiliser tableau de hashage :

```
%hash = ( foo => "un",
          bar => 2, );
print $hash{'bar'};
$clef = 'clef'; $hash{$clef};
```

Attention au $

```
%hash;
$hash{baz};
```

hash : exemple

Utiliser tableau de hashage :

```
%hash = ( foo => "un",
          bar => 2, );
print $hash{'bar'};
$clef = 'clef'; $hash{$clef};
```

Attention au $

```
%hash;
$hash{baz};
```

keys : renvois les clefs

```
keys(%hash);
```

2 Variables
- Variables de Perl
- Scalaire
- Les tableaux
- Les tables de hashage
- Les listes
- les variables magiques

Comprendre les listes

Perl aplati les listes !

Le concept est simple, mais redoutable :

Comprendre les listes

Perl aplati les listes !

Le concept est simple, mais redoutable :

Prenons deux tableaux de trois entrées :

```
@tab1 = (1, 2, 3);
@tab2 = (4, 5, 6);
```

Comprendre les listes

Perl aplati les listes !

Le concept est simple, mais redoutable :

Prenons deux tableaux de trois entrées :

```
@tab1 = (1, 2, 3);
@tab2 = (4, 5, 6);
```

```
($un) = @tab1;
```

On copie la première entrée du tableau dans $un

Comprendre les listes

Perl aplati les listes !

Le concept est simple, mais redoutable :

Prenons deux tableaux de trois entrées :

```
@tab1 = (1, 2, 3);
@tab2 = (4, 5, 6);
```

```
($un, @reste) = @tab1;
```

pareil, mais la fin du tableaux va dans @reste

Comprendre les listes

Perl aplati les listes !

Le concept est simple, mais redoutable :

Prenons deux tableaux de trois entrées :

```
@tab1 = (1, 2, 3);
@tab2 = (4, 5, 6);
```

```
($un, @reste) = (@tab1, @tab2);
```

@tab1 et @tab2 ne forme plus qu'un lors de la copie

Comprendre les listes

Perl aplati les listes !

Le concept est simple, mais redoutable :

Prenons deux tableaux de trois entrées :

```
@tab1 = (1, 2, 3);
@tab2 = (4, 5, 6);
```

```
(undef, $un, @reste) = (@tab1, @tab2);
```

la première entrée est ignoré (undef)

Comprendre les listes : les hash

'=>' est un séparateur comme la ', ', donc c'est une liste :

```
%hash = (un => 1, deux => 2);
%hash = (un, 1, deux, 2);
@tab = %hash; %hash = @tableau;
```

`@tab` contient "un", 1...

Comprendre les listes : les hash

'=>' est un séparateur comme la ', ', donc c'est une liste :

```
%hash = (un => 1, deux => 2);
%hash = (un, 1, deux, 2);
@tab = %hash; %hash = @tableau;
```

`@tab` contient "un", 1...

hash vers hash

```
%unhash = (%hash2, %hash3);
```

`%hash` contient 1 => 2, 3 => 4...

Comprendre les listes : les hash

'=>' est un séparateur comme la ',', donc c'est une liste :

```
%hash = (un => 1, deux => 2);
%hash = (un, 1, deux, 2);
@tab = %hash; %hash = @tableau;
```

`@tab` contient "un", 1...

hash vers hash

```
%unhash = (%hash2, %hash3);
```

`%hash` contient 1 => 2, 3 => 4...

Les clefs dupliquées sont ecrasées, la dernière gagne !

2 Variables
- Variables de Perl
- Scalaire
- Les tableaux
- Les tables de hashage
- Les listes
- les variables magiques

Elles existent dans tout programme perl.

Les principales à connaitre :

Elles existent dans tout programme perl.

Les principales à connaitre :

`$_` stockage par défaut

Elles existent dans tout programme perl.

Les principales à connaitre :

`$_`	stockage par défaut
`@_`	paramètres des subroutines

Elles existent dans tout programme perl.

Les principales à connaitre :

`$_`	stockage par défaut
`@_`	paramètres des subroutines
`@ARGV`	les arguments passés aux programmes

Elles existent dans tout programme perl.

Les principales à connaitre :

`$_`	stockage par défaut
`@_`	paramètres des subroutines
`@ARGV`	les arguments passés aux programmes
`$a` et `$b`	utilisé par sort

Elles existent dans tout programme perl.

Les principales à connaitre :

`$_`	stockage par défaut
`@_`	paramètres des subroutines
`@ARGV`	les arguments passés aux programmes
`$a` et `$b`	utilisé par sort

Attention à $_

`$_` permet d'écrire du code succin, mais n'augmente pas la visibilité :

```
$entier = int;
```

Plan

1. Presentation
2. Variables
3. Syntaxe
   - Syntaxe générale
   - Opérateurs
   - Controles d'execution
   - Fonctions et subroutines
   - Traitement sur les listes
   - Déclarer ses variables
4. Chaines de caractères
5. Variables bis

3 Syntaxe
- Syntaxe générale
- Opérateurs
- Controles d'execution
- Fonctions et subroutines
- Traitement sur les listes
- Déclarer ses variables

Grandes lignes

- les espaces sépare les mots

Grandes lignes

- les espaces sépare les mots
- le '`;`' sépare les instructions

Grandes lignes

- les espaces sépare les mots
- le '`;`' sépare les instructions
- le '`,`' sépare les valeurs

Grandes lignes

- les espaces sépare les mots
- le '`;`' sépare les instructions
- le '`,`' sépare les valeurs
- `(` `)` limite une liste

Grandes lignes

- les espaces sépare les mots
- le '`;`' sépare les instructions
- le '`,`' sépare les valeurs
- `(` `)` limite une liste
- `{` `}` limite un bloc

Grandes lignes

- les espaces sépare les mots
- le '`;`' sépare les instructions
- le '`,`' sépare les valeurs
- `(` `)` limite une liste
- `{` `}` limite un bloc
- tout ce qui suit un `#` est un commentaire

Grandes lignes

- les espaces sépare les mots
- le '`;`' sépare les instructions
- le '`,`' sépare les valeurs
- `(` `)` limite une liste
- `{` `}` limite un bloc
- tout ce qui suit un `#` est un commentaire
- `__END__` indique la fin des instructions

Grandes lignes

- les espaces sépare les mots
- le ';' sépare les instructions
- le ',' sépare les valeurs
- ( ) limite une liste
- { } limite un bloc
- tout ce qui suit un # est un commentaire
- __END__ indique la fin des instructions

Tout est permis, y compris apporter de la lisibilité à votre code :

```
      push # création du tableau
(         @tableau
  , $valeur,     $valeur2     );
```

Structure classique d'un script

un shell bang :

Structure classique d'un script

un shell bang :

```
#!env perl
```

Structure classique d'un script

un shell bang :

```
#!env perl
```

charger les modules dont on a besoin :

Structure classique d'un script

un shell bang :

```
#!env perl
```

charger les modules dont on a besoin :

```
use strict;
use warnings;
```

Structure classique d'un script

un shell bang :

```
#!env perl
```

charger les modules dont on a besoin :

```
use strict;
use warnings;
```

arrive le code :

Structure classique d'un script

un shell bang :

```
#!env perl
```

charger les modules dont on a besoin :

```
use strict;
use warnings;
```

arrive le code :

```
while (<STDIN>) {
    s/ //g;
    print $_;
}
```

Structure classique d'un script

```
#!env perl


use strict;
use warnings;


while (<STDIN>) {
    s/ //g;
    print $_;
}

__END__
```

3 Syntaxe
- Syntaxe générale
- Opérateurs
- Controles d'execution
- Fonctions et subroutines
- Traitement sur les listes
- Déclarer ses variables

Opérateur

Une courte liste

+ − * /	devinez !
%	modulo

Opérateur

Une courte liste

`+ - * /`	devinez !
`%`	modulo
`&& ||`	et/ou logique
`and or`	et/ou
`? :`	if then else

Opérateur

Une courte liste

`+ - * /`	devinez !
`%`	modulo
`&& ||`	et/ou logique
`and or`	et/ou
`? :`	if then else
`.`	concaténation de chaine

Opérateur

Une courte liste

`+ - * /`	devinez !
`%`	modulo
`&& ||`	et/ou logique
`and or`	et/ou
`? :`	if then else
`.`	concaténation de chaine
`& |`	et/ou logique (bit)
`<< >>`	décalage de bit

```
man perlop
```

3 Syntaxe
- Syntaxe générale
- Opérateurs
- Controles d'execution
- Fonctions et subroutines
- Traitement sur les listes
- Déclarer ses variables

if

Bah... if quoi...

```
if ($var == 0) {
    print "var vaut 0\n";
}
```

if

Bah... if quoi...

```
if ($var == 0) {
    print "var vaut 0\n";
}





 else {
    print "var vaut $var\n";
}
```

if

Bah... if quoi...

```
if ($var == 0) {
    print "var vaut 0\n";
}

 elsif ($var == 1) {
    print "var vaut 1\n";
}

 else {
    print "var vaut $var\n";
}
```

if

Bah... if quoi...

```
if ($var == 0) {
    print "var vaut 0\n";
}

 elsif ($var == 1) {
    print "var vaut 1\n";
}

 else {
    print "var vaut $var\n";
}
```

Forme suffixée :

```
print "var vaut 0\n" if ($var == 0);
```

while

Boucle tant qu'une condition est vraie :

```
while ($i < 100) {
    print "$i\n";
    $i++;
}
```

while

Boucle tant qu'une condition est vraie :

```
while ($i < 100) {
    print "$i\n";
    $i++;
}

do {
    print "$i\n";
    $i++;
} while($i < 100);
```

while

Boucle tant qu'une condition est vraie :

```
while ($i < 100) {
    print "$i\n";
    $i++;
}

do {
    print "$i\n";
    $i++;
} while($i < 100);
```

Forme suffixée :

```
$i++ while($i < 100);
```

foreach

Boucle pour chaque élément d'une liste :

```
foreach $elem (@tableau) {
    print "$elem\n";
}
```

Très pratique, cela évite de s'embeter avec les indices du tableau.

foreach

Boucle **pour chaque** élément d'une liste :

```
foreach $elem (@tableau) {
    print "$elem\n";
}
```

Très pratique, cela évite de s'embeter avec les indices du tableau.

Forme suffixée :

```
print "$_\n" foreach(@tableau);
```

Sortez moi de là

next fait passer à la prochaine itération d'une boucle :

```
foreach $val (@tableau) {
    next if(!$val);
    print "$val\n";
}
```

Sortez moi de là

next fait passer à la prochaine itération d'une boucle :

```
foreach $val (@tableau) {
    next if(!$val);
    print "$val\n";
}
```

last quitte une boucle :

```
foreach $val (@tableau) {
    if ($val > 1000) {
        print "$val est trop haut, arret\n";
        last;
    }
    print "$val\n";
}
```

3 Syntaxe
- Syntaxe générale
- Opérateurs
- Controles d'execution
- Fonctions et subroutines
- Traitement sur les listes
- Déclarer ses variables

Fonctions

Déclarer une fonction :

```
sub fonction {
    print "Hello World\n";
}
```

Fonctions

Déclarer une fonction :

```
sub fonction {
    print "Hello World\n";
}
```

Utilisation

```
fonction();

fonction;
```

Fonctions

Déclarer une fonction :

```
sub fonction {
    print "Hello World\n";
}
```

Utilisation

```
fonction();

fonction;
```

Sub anonyme, plus rarement utilisé :

```
my $sub = sub { };
$sub->();
```

Fonction : passage d'arguments

les arguments sont passé dans @_

Fonction : passage d'arguments

les arguments sont passé dans @_

On passe 1 arguments

```
sub affiche {
    ($arg) = @_;
    print "$arg\n";
}

affiche("foo");
```

Fonction : passage d'arguments

les arguments sont passé dans @_

On n arguments (ici 2)

```
sub affiche {
    ($arg, $arg2) = @_;
    print "$arg\n";
}

affiche("foo", "bar");
```

Fonction : passage d'arguments

les arguments sont passé dans @_

Via un hash :

- arguments nommés
- arguments optionnels

```
sub affiche {
    (%hash) = @_;
    print $hash{key1};
}

affiche(key1 => "val1");
```

Fonction : passage d'arguments

les arguments sont passé dans @_

on rends le premier arguments obligatoire, les autres sont optionnels

```
sub affiche {
    ($arg, %hash) = @_;
    print "$arg\n";
}

affiche("foo", key1 => "val1");
```

Retours des valeurs

la dernière valeur est retournée

Retours des valeurs

la dernière valeur est retournée

On renvoit explicitement $res

```
sub addition {
    ($un, $deux) = @_;
    $res = $un + $deux;
    return $res
}

print addition(2, 4);
```

Retours des valeurs

la dernière valeur est retournée

On renvoit implicitement $res (dernière valeur)

```
sub addition {
    ($un, $deux) = @_;
    $res = $un + $deux;
    $res
}

print addition(2, 4);
```

Retours des valeurs

la dernière valeur est retournée

On renvoit explicitement le résultat

```
sub addition {
    ($un, $deux) = @_;
    return $un + $deux;

}

print addition(2, 4);
```

Retours des valeurs

la dernière valeur est retournée

le resultat est la dernière valeur...

```
sub addition {
    ($un, $deux) = @_;
    $un + $deux;

}

print addition(2, 4);
```

3 Syntaxe
- Syntaxe générale
- Opérateurs
- Controles d'execution
- Fonctions et subroutines
- Traitement sur les listes
- Déclarer ses variables

sort

Fonction de tri

Evalue la comparaison entre `$a` et `$b`
Selon le résultat, l'une ou l'autre passe devant

sort

Fonction de tri

Evalue la comparaison entre `$a` et `$b`
Selon le résultat, l'une ou l'autre passe devant

```
my @tableau = ('010', '1', '12');
```

sort

Fonction de tri

Evalue la comparaison entre `$a` et `$b`
Selon le résultat, l'une ou l'autre passe devant

```
my @tableau = ('010', '1', '12');
```

tri numérique

```
sort { $a <=> $b } @tableau;
```

sort

Fonction de tri

Evalue la comparaison entre `$a` et `$b`
Selon le résultat, l'une ou l'autre passe devant

```
my @tableau = ('010', '1', '12');
```

tri numérique

```
sort { $a <=> $b } @tableau;
```

tri alphabétique

```
sort { $a cmp $b } @tableau;
```

sort

Fonction de tri

Evalue la comparaison entre `$a` et `$b`
Selon le résultat, l'une ou l'autre passe devant

```
my @tableau = ('010', '1', '12');
```

tri numérique

```
sort { $a <=> $b } @tableau;
```

tri alphabétique

```
sort { $a cmp $b } @tableau;
```

tri numérique inverse

```
sort { $b <=> $a } @tableau;
```

grep

Fonction de filtre

Revois chaque valeur placée dans `$_` si la condition renvoyée est vraie

grep

Fonction de filtre

Revois chaque valeur placée dans `$_` si la condition renvoyée est vraie

```
my @tableau = ('010', '1', '12');
```

grep

Fonction de filtre

Revois chaque valeur placée dans `$_` si la condition renvoyée est vraie

```
my @tableau = ('010', '1', '12');
```

valeur vraie

```
grep { $_ } @tableau;
```

grep

Fonction de filtre

Revois chaque valeur placée dans `$_` si la condition renvoyée est vraie

```
my @tableau = ('010', '1', '12');
```

valeur vraie

```
grep { $_ } @tableau;
```

avec une condition

```
grep { $_ > 10 } @tableau;
```

grep

Fonction de filtre

Revois chaque valeur placée dans `$_` si la condition renvoyée est vraie

```
my @tableau = ('010', '1', '12');
```

valeur vraie

```
grep { $_ } @tableau;
```

avec une condition

```
grep { $_ > 10 } @tableau;
```

sans utiliser $_

```
my $vrai = 1
grep { $vrai } @tableau;
```

map

mappage des valeurs

Remplace chaque valeur par l'évaluation retournée
Chaque valeur est placé dans `$_`

map

mappage des valeurs

Remplace chaque valeur par l'évaluation retournée
Chaque valeur est placé dans `$_`

```
my @tableau = ('010', '1', '12');
```

map

mappage des valeurs

Remplace chaque valeur par l'évaluation retournée
Chaque valeur est placé dans `$_`

```
my @tableau = ('010', '1', '12');
```

renvois chaque valeur + 1

```
map { $_ + 1 } @tableau;
```

map

mappage des valeurs

Remplace chaque valeur par l'évaluation retournée
Chaque valeur est placé dans `$_`

```
my @tableau = ('010', '1', '12');
```

renvois chaque valeur + 1

```
map { $_ + 1 } @tableau;
```

renvois chaque valeur et chaque valeur + 1

```
map { $_, $_ + 1 } @tableau;
```

Combinaison pour la forme

humpff...

```
my @communes =

    map { ucfirst($_) }

    sort { $a cmp $b }

    grep { substr($_, 0, 1) eq 'm' }

    ('nantes',
     'mont de marsan', 'amiens', 'nancy', 'dijon',
     'bordeaux', 'metz', bordeaux', 'bordeaux', );
```

Combinaison pour la forme

humpff...

```
my @communes =

    map { ucfirst($_) }

    sort { $a cmp $b }

    grep { substr($_, 0, 1) eq 'm' }
    # une liste de villes
    ('nantes',
     'mont de marsan', 'amiens', 'nancy', 'dijon',
     'bordeaux', 'metz', bordeaux', 'bordeaux', );
```

Combinaison pour la forme

humpff...

```
my @communes =

    map { ucfirst($_) }

    sort { $a cmp $b }
    # filtre : première lettre est un m
    grep { substr($_, 0, 1) eq 'm' }
    # une liste de villes
    ('nantes',
     'mont de marsan', 'amiens', 'nancy', 'dijon',
     'bordeaux', 'metz', bordeaux', 'bordeaux', );
```

Combinaison pour la forme

humpff...

```
my @communes =

    map { ucfirst($_) }
    # tri : ordre alphabétique
    sort { $a cmp $b }
    # filtre : première lettre est un m
    grep { substr($_, 0, 1) eq 'm' }
    # une liste de villes
    ('nantes',
     'mont de marsan', 'amiens', 'nancy', 'dijon',
     'bordeaux', 'metz', bordeaux', 'bordeaux', );
```

Combinaison pour la forme

humpff...

```
my @communes =
    # première lettre en majuscule
    map { ucfirst($_) }
    # tri : ordre alphabétique
    sort { $a cmp $b }
    # filtre : première lettre est un m
    grep { substr($_, 0, 1) eq 'm' }
    # une liste de villes
    ('nantes',
     'mont de marsan', 'amiens', 'nancy', 'dijon',
     'bordeaux', 'metz', bordeaux', 'bordeaux', );
```

3 Syntaxe
- Syntaxe générale
- Opérateurs
- Controles d'execution
- Fonctions et subroutines
- Traitement sur les listes
- Déclarer ses variables

my and compagny

Perl est très laxiste :

```
$latitude = 5;
print "$latitud\n";
```

my and compagny

Perl est très laxiste :

Voici un code qui fonctionne, mais pas comme on veut...

```
$latitude = 5;
print "$latitud\n";
```

my and compagny

Perl est très laxiste :

Deux moyens de forcer les décalarations :

- **use strict ;** force la declaration des variables

```
$latitude = 5;
print "$latitud\n";
```

my and compagny

Perl est très laxiste :

Deux moyens de forcer les décalarations :

- **use strict ;** force la declaration des variables
- **use warnings ;** produit des warnings sur les actions douteuses

```
$latitude = 5;
print "$latitud\n";
```

my and compagny

Perl est très laxiste :

Deux moyens de forcer les décalarations :

- **use strict ;** force la declaration des variables
- **use warnings ;** produit des warnings sur les actions douteuses

l'erreur :

Global symbol "$latitude" requires explicit package

```
use strict;

$latitude = 5;
print "$latitud\n";
```

my and compagny

Perl est très laxiste :

Deux moyens de forcer les décalarations :

- **use strict ;** force la declaration des variables
- **use warnings ;** produit des warnings sur les actions douteuses

my sert à déclarer une variable

```
use strict;

my $latitude = 5;
print "$latitud\n";
```

my and compagny

Perl est très laxiste :

Deux moyens de forcer les décalarations :

- **use strict ;** force la declaration des variables
- **use warnings ;** produit des warnings sur les actions douteuses

my sert à déclarer une variable

l'erreur :

Global symbol "$latitud" requires explicit package

```
use strict;

my $latitude = 5;
print "$latitud\n";
```

my and compagny

Perl est très laxiste :

Deux moyens de forcer les décalarations :

- **use strict ;** force la declaration des variables
- **use warnings ;** produit des warnings sur les actions douteuses

my sert à déclarer une variable
Voilà qui est parfait !

```
use strict;

my $latitude = 5;
print "$latitude\n";
```

The use strict rules

J'ai bien pris note que :

The use strict rules

J'ai bien pris note que :

J'utiliserais au moins *use strict*

The use strict rules

J'ai bien pris note que :

J'utiliserais au moins *use strict*
J'utiliserais **toujours** au moins *use strict*

The use strict rules

J'ai bien pris note que :

J'utiliserais au moins *use strict*
J'utiliserais **toujours** au moins *use strict*
Jamais je ne dérangerais un gourou Perl si je n'ai pas utilisé *use strict*

Portée des variables

`$var` différent de `@var` différent de `%var` !

Portée des variables

`$var` différent de `@var` différent de `%var` !

les règles

Sans `use strict;` ni `my` Perl fait au mieux.

Portée des variables

`$var` différent de `@var` différent de `%var` !

les règles

Sans `use strict;` ni `my` Perl fait au mieux.
Les déclarations se font n'importe où, mais avant leur utilisation.

Portée des variables

`$var` différent de `@var` différent de `%var` !

les règles

Sans `use strict;` ni `my` Perl fait au mieux.
Les déclarations se font n'importe où, mais avant leur utilisation.

example :

```
use strict;

my $var = 1
sub mafonction {
    print $var;
    my $fvar = 3;
}
```

Portée des variables

`$var` différent de `@var` différent de `%var` !

les règles

Sans `use strict;` ni `my` Perl fait au mieux.
Les déclarations se font n'importe où, mais avant leur utilisation.

example :

```
use strict;
mafonction();
my $var = 1
sub mafonction {
    print $var;
    my $fvar = 3;
}
```

boucles et blocs

limiter la portée en utilisant un bloc

```
my $globalvar = 1;
{
    my $blockvar = 2;
}
```

boucles et blocs

limiter la portée en utilisant un bloc

```
my $globalvar = 1;
{
    my $blockvar = 2;
}
```

Déclaration dans les boucles

```
my @tableau = (1, 2, 3, 4);

foreach my $item (@tableau) {
    print "$item\n";
}
```

Plan

1. Presentation
2. Variables
3. Syntaxe
4. Chaines de caractères
   - quote
   - RegExp
5. Variables bis
6. Fin

4 Chaines de caractères
- quote
- RegExp

Opérateur et délimiteurs

faire	interpollée	non interpolée
une chaine	" "	' '
une chaine	q//	qq//
une liste		qw//

Opérateur et délimiteurs

faire	interpollée	non interpolée
une chaine	`" "`	`' '`
une chaine	`q//`	`qq//`
une liste		qw//

Délimiteurs des opérateurs :

- n'importe quels caractères
- les couples logiques `()`, `[]`, `{}`

Opérateur et délimiteurs

faire	interpollée	non interpolée
une chaine	`" "`	`' '`
une chaine	`q//`	`qq//`
une liste		qw//

Délimiteurs des opérateurs :

- n'importe quels caractères
- les couples logiques `()`, `[]`, `{}`

Exemple : `q//`, `q;;`, `q[]`, `q{}`

quote : exemple

Chaine interpolées :

```
my $var = "UMR7620";

print "nous somme \"$var\"";
print qq!nous somme "$var"!;
```

quote : exemple

Chaine interpolées :

```
my $var = "UMR7620";

print "nous somme \"$var\"";
print qq!nous somme "$var"!;
```

Chaine non interpolées :

```
my $var = 'UMR7620';

print '$var est une variable';
print q{'$var' est une variable};
```

quote : exemple

Chaine interpolées :

```
my $var = "UMR7620";

print "nous somme \"$var\"";
print qq!nous somme "$var"!;
```

Chaine non interpolées :

```
my $var = 'UMR7620';

print '$var est une variable';
print q{'$var' est une variable};
```

une liste

```
my @tableau = qw(une liste de mots);
```

quote quote

Pour les longues chaines :

```
my $var = 42;

print <<FIN_DU_TEXTE;
La réponse à la question
universelle est $var.
FIN_DU_TEXTE
```

4 Chaines de caractères

- quote
- RegExp

Perl réinvente les RegExp

Comment ne pas parler des Expéssions Régulières ?

Elles sont

- puissantes
- très étendues
- naturelles en perl

Perl Compatible Regular Expression

Il y a même une librairie qui émule les expression régulière de PERL. Utilisé par

- grep (et oui)
- php
- kde
- ...

RegExp : les bases

Regarde si une chaine correspond à un motif

le motif peut contenir :

- . 1 caractère

RegExp : les bases

Regarde si une chaine correspond à un motif

le motif peut contenir :

.	1 caractère
*	0 ou plusieurs fois
+	1 ou plusieurs fois
?	1 fois

RegExp : les bases

Regarde si une chaine correspond à un motif

le motif peut contenir :

.	1 caractère
*	0 ou plusieurs fois
+	1 ou plusieurs fois
?	1 fois
\	echape un caractère magique

RegExp : les bases

Regarde si une chaine correspond à un motif

le motif peut contenir :

.	1 caractère
*	0 ou plusieurs fois
+	1 ou plusieurs fois
?	1 fois
\	echape un caractère magique
^	début de ligne
$	fin de ligne

RegExp : les bases

Regarde si une chaine correspond à un motif

le motif peut contenir :

`.`	1 caractère
`*`	0 ou plusieurs fois
`+`	1 ou plusieurs fois
`?`	1 fois
`\`	echape un caractère magique
`^`	début de ligne
`$`	fin de ligne
`( )`	forme un groupe

Anatomie

L'expression régulière :

```
$var                    ;
$var                    ;
$var                                ;
```

Anatomie

L'expression régulière :

```
$var =~                ;
$var =~                ;
$var =~                          ;
```

- un opérateur de comparaison
  - =~ : correspond
  - !~ : ne correspond pas

Anatomie

L'expression régulière :

```
$var =~  m          ;
$var =~             ; # séparateur / => opérateur m
$var =~  s                        ;
```

- un opérateur de comparaison
  - =~ : correspond
  - !~ : ne correspond pas
- une action
  - **m** (match, le défaut)
  - **s** (substitute)

Anatomie

L'expression régulière :

```
$var =~  m/     /  ;
$var =~   /     /  ; # séparateur / => opérateur m
$var =~  s/     /            /  ;
```

- un opérateur de comparaison
  - =~ : correspond
  - !~ : ne correspond pas
- une action
  - **m** (match, le défaut)
  - **s** (substitute)
- des séparateurs (au choix)

Anatomie

L'expression régulière :

```
$var =~  m/motif/ ;
$var =~   /motif/ ; # séparateur / => opérateur m
$var =~  s/motif/remplacement/ ;
```

- un opérateur de comparaison
  - =~ : correspond
  - !~ : ne correspond pas
- une action
  - **m** (match, le défaut)
  - **s** (substitute)
- des séparateurs (au choix)
- le texte recherché (et le remplacement)

Anatomie

L'expression régulière :

```
$var =~  m/motif/i;
$var =~   /motif/i; # séparateur / => opérateur m
$var =~  s/motif/remplacement/g;
```

- un opérateur de comparaison
  - =~ : correspond
  - !~ : ne correspond pas
- une action
  - **m** (match, le défaut)
  - **s** (substitute)
- des séparateurs (au choix)
- le texte recherché (et le remplacement)
- éventuellement un modificateur

RegExp : exemple dans perl

```
my @list = ('UMR7620', 'umr8639');

foreach my $umr (@list) {
    my ($number) = $umr =~ /^umr(.*)$/;
    if ($number)  {
        print "Unité mixte $number\n";
    }
}
```

RegExp : exemple dans perl

```
my @list = ('UMR7620', 'umr8639');

foreach my $umr (@list) {
    my ($number) = $umr =~ /^umr(.*)$/;
    if ($number)  {
        print "Unité mixte $number\n";
    }
}
```

la doc !

- man perlre
- man perlretut

Plan

1. Presentation
2. Variables
3. Syntaxe
4. Chaines de caractères
5. Variables bis
   - Références
   - les descripteurs de fichier
6. Fin

5 Variables bis

- Références
- les descripteurs de fichier

Référence : Concept

Quoi ça ?

C'est une valeur scalaire...

Référence : Concept

Quoi ça ?

C'est une valeur scalaire...
qui fait référence à un scalaire, un tableau ou un hash.

Référence : Concept

Quoi ça ?

C'est une valeur scalaire...
qui fait référence à un scalaire, un tableau ou un hash.

- elle contient l'adresse mémoire de l'élément pointé

Référence : Concept

Quoi ça ?

C'est une valeur scalaire...
qui fait référence à un scalaire, un tableau ou un hash.

- elle contient l'adresse mémoire de l'élément pointé
- plusieurs références peuvent pointer la même variable

Référence : Concept

Quoi ça ?

C'est une valeur scalaire...
qui fait référence à un scalaire, un tableau ou un hash.

- elle contient l'adresse mémoire de l'élément pointé
- plusieurs références peuvent pointer la même variable
- on peut faire des tableaux de références...

Référence : Concept

Quoi ça ?

C'est une valeur scalaire...
qui fait référence à un scalaire, un tableau ou un hash.

- elle contient l'adresse mémoire de l'élément pointé
- plusieurs références peuvent pointer la même variable
- on peut faire des tableaux de références...

Ce n'est pas un pointeur au sens C du terme

`$ref++;` ne donnera rien de bien !!!!
On fait du perl, on ne joue pas avec les octets de la mémoire.

Créer des références

Renvoyer la référence de

\ pour renvoyer un référence :

```
my $ref = \$scalaire;
my $ref = \@tableau;
my $ref = \%hash;
```

Créer des références

Renvoyer la référence de

\ pour renvoyer un référence :

```
my $ref = \$scalaire;
my $ref = \@tableau;
my $ref = \%hash;
```

Créer une référence :

Type de variable	code
tableau	`[ 'val1', 'val2' ]`
hash	`{ key => val }`

Utiliser les références

prenons une référence

```
$ref = { key => 'val' }; print $ref;
HASH(0x8bca880)
```

Utiliser les références

prenons une référence

```
$ref = { key => 'val' }; print $ref;
HASH(0x8bca880)
```

Accès aux variable

type	variables	éléments de
scalaire	`$$var`	
tableau	`@{$ref}`	`$ref->[0]`
hash	`%{$ref}`	`$ref->{'key'}`

Utiliser les références

prenons une référence

```
$ref = { key => 'val' }; print $ref;
HASH(0x8bca880)
```

Accès aux variable

type	variables	éléments de
scalaire	`$$var`	
tableau	`@{$ref}`	`$ref->[0]`
hash	`%{$ref}`	`$ref->{'key'}`

```
foreach my $key (keys %$ref) {
    print "$ref->{$key}\n";
}
```

Pourquoi les références ?

Perl aplati les listes, remember ?

Pourquoi les références ?

Perl aplati les listes, remember ?

```
my @tab1 = (1,2); my @tab2 = (3,4);
my @tab = (@tab1, @tab2);
```

`@tab` contient 4 éléments...

Pourquoi les références ?

Perl aplati les listes, remember ?

```
my @tab1 = (1,2); my @tab2 = (3,4);
my @tab = (@tab1, @tab2);
```

`@tab` contient 4 éléments...

par contre :

```
my @tab1 = (1,2); my @tab2 = (3,4);
my @tab = (\@tab1, \@tab2);
```

`@tab` contient 2 éléments qui réferencent des tableaux.

Pourquoi les références ?

Perl aplati les listes, remember ?

```
my @tab1 = (1,2); my @tab2 = (3,4);
my @tab = (@tab1, @tab2);
```

`@tab` contient 4 éléments...

par contre :

```
my @tab1 = (1,2); my @tab2 = (3,4);
my @tab = (\@tab1, \@tab2);
```

`@tab` contient 2 éléments qui réferencent des tableaux.

On accède aux valeurs :

```
$tab[0]->[0];
$tab[0][0];
```

Références : les pièges

Déférencer un undef :

```
perl -we 'my $ref; print @{ $ref }'
Use of uninitialized value $ref in
array dereference at -e line 1.
```

Références : les pièges

Déférencer un undef :

```
perl -we 'my $ref; print @{ $ref }'
Use of uninitialized value $ref in
array dereference at -e line 1.
```

Attention au type de variable référencée :

```
perl -we 'my $ref = {}; print @{ $ref }'
Not an ARRAY reference at -e line 1.
```

Références : les pièges

Déférencer un undef :

```
perl -we 'my $ref; print @{ $ref }'
Use of uninitialized value $ref in
array dereference at -e line 1.
```

Attention au type de variable référencée :

```
perl -we 'my $ref = {}; print @{ $ref }'
Not an ARRAY reference at -e line 1.
```

Attention au structure complexes :

```
$ref->[0][2]{daleks}[0]{episode}
    {$ennemi}{histoire}[0];
```

5 Variables bis

- Références
- les descripteurs de fichier

Fichiers : Version simple

Ouvrir un fichier en lecture :

```
open(my $handle, '<', 'fichier');
```

Fichiers : Version simple

Ouvrir un fichier en lecture :

```
open(my $handle, '<', 'fichier');
```

Lire une ligne :

```
my $line = <$handle>;
chomp($line); # strip cariage return
```

Fichiers : Version simple

Ouvrir un fichier en lecture :

```
open(my $handle, '<', 'fichier');
```

Lire une ligne :

```
my $line = <$handle>;
chomp($line); # strip cariage return
```

Fermer le fichier :

```
close($handle);
```

Fichiers : Version simple

Ouvrir un fichier en lecture :

```
open(my $handle, '<', 'fichier');
```

Lire une ligne :

```
my $line = <$handle>;
chomp($line); # strip cariage return
```

Fermer le fichier :

```
close($handle);
```

Ancienne méthode (perl < 5.6)

```
open(HANDLE, '<', 'fichier');
my $line = <HANDLE>;
close(HANDLE);
```

Fichier : version simple (suite)

Ecrire

```
open(my $handle, '>', 'fichier');
print $handle "Exterminate !\n";
close($handle);
```

Fichier : version simple (suite)

Ecrire

```
open(my $handle, '>', 'fichier');
print $handle "Exterminate !\n";
close($handle);
```

Attention à print

pas de , après le file handle

Fichier : version simple (suite)

Ecrire

```
open(my $handle, '>', 'fichier');
print $handle "Exterminate !\n";
close($handle);
```

Attention à print

pas de , après le file handle

File handle par défaut

- STDERR (sortie d'erreur)
- STDOUT (sortie standard)
- STDIN (entrée standard)

Fichiers : Version compliquée

haut niveau

- open()/close()
- read()/print()

Fichiers : Version compliquée

haut niveau

- open()/close()
- read()/print()

bas niveau

- sysopen()/sysclose()
- sysread()/syswrite()

Fichiers : Version compliquée

haut niveau

- open()/close()
- read()/print()

bas niveau

- sysopen()/sysclose()
- sysread()/syswrite()

Glob !

Pour info, un filehandle est une variable de type **GLOB**.

Fichiers : Version compliquée

haut niveau

- open()/close()
- read()/print()

bas niveau

- sysopen()/sysclose()
- sysread()/syswrite()

Glob !

Pour info, un filehandle est une variable de type **GLOB**.

Si vraiment vous voulez en savoir plus : man

- perlfunc
- perlopentut

Plan

1. Presentation
2. Variables
3. Syntaxe
4. Chaines de caractères
5. Variables bis
6. Fin

La doc !

- `perldoc`
- `man`

La doc !

- `perldoc`
- `man`

Les références

perlfunc	functions de base

La doc !

- `perldoc`
- `man`

Les références

perlfunc	functions de base
perlop	les opérateurs

La doc !

- `perldoc`
- `man`

Les références

perlfunc	functions de base
perlop	les opérateurs
perlvar	les variables de perl

La doc !

- `perldoc`
- `man`

Les références

perlfunc	functions de base
perlop	les opérateurs
perlvar	les variables de perl
perldata	utilisation des variables

La doc !

- `perldoc`
- `man`

Les références

perlfunc	functions de base
perlop	les opérateurs
perlvar	les variables de perl
perldata	utilisation des variables
perlref	les références
perlreftut	

La doc !

- `perldoc`
- `man`

Les références

perlfunc	functions de base
perlop	les opérateurs
perlvar	les variables de perl
perldata	utilisation des variables
perlref	les références
perlreftut	
perlopentut	tout sur open

La doc !

- `perldoc`
- `man`

Les références

perlfunc	functions de base
perlop	les opérateurs
perlvar	les variables de perl
perldata	utilisation des variables
perlref	les références
perlreftut	
perlopentut	tout sur open

La communauté

Perl

http://www.perl.org/
http://cpan.perl.org/

La communauté

Perl

http://www.perl.org/
http://cpan.perl.org/

les Mongueurs

Association loi 1901
http://www.mongueurs.net/
http://paris.mongueurs.net/ (Paris)

Vous les retrouverez dans "Linux Magazine" et aux Journées Perl

Questions ?