


4^{ème} séance : bibliothèques Perl et programmation objet

- bibliothèques : web requête/analyse, courriel, SSH, LDAP, XML, SVG, PDF, ODF, YAML, JSON ...

```
$p=Net::Ping->new; if($p->ping($host)){...}
$ftp=Net::FTP->new($host); $ftp->get($file);
$ua=LWP::UserAgent->new; $ua->proxy(...);
$res=$ua->request(HTTP::Request->new(GET=>$url)); #HTTP::Response
```


- POO native : composition, héritage, exception, surcharge opérateur ...
 accesseur polyvalent : `$pers->age(12); $age=$pers->age();`
- POO Moose : concepts innovants (Perl 6), typage, héritage, trigger
 - délégation d'une méthode à un attribut (ex. structure Perl)


```
has stationne=>(isa=>'ArrayRef[Voiture]',handles=>{garer=>'push'});
$parking->garer($voiture); # Pas de méthode à écrire
```
 - augmentation de méthode : pré/post-traitement (*per* lors héritage/rôle)

```
before garer => sub { # Vérification s'il y a de la place }
```
 - concept de **rôle** : mi-chemin entre interface et classe abstraite Java

5^{ème} séance : outils Perl pour le système

- Bonnes pratiques *Toujours coder comme si le gars qui va finir par maintenir votre code est un psychopathe violent qui sait où vous habitez.* – Rick Osborne
- Appels systèmes : fichiers, processus ...
- Sockets / Threads (exclusion mutuelle aisée)
- Courriels : réception, envoi, attachement, analyse

```
$msg=MIME::Lite->new(To=>'a@dres.se',Subject=>'Ok');
$msg->attach(Path=>'image.jpg'); $msg->send();
```



- Gestion du temps : fuseaux horaires, analyse de date


```
$d->parse('1st Monday June 1992'); $d->parse('12:30 Dec 12th 1880');
```
- Gestion des paramètres de la ligne de commande

```
prog -l --size=14 --coord 6.4 8.1 --rgbcolor 255 0 0 fichiers
```
- Filtres courts : utiliser Perl en shell (rouge et non Montrouge)

```
perl -i 'save/*.bak' -pe 's/\brouge\b/blanc/' *.txt
```

6^{ème} séance : web dynamique, bases de données

- Web dynamique
 génération formulaire, cookies, upload ...
- Bases de données : requêtes, transactions ...
 Sécurité : éviter les injection SQL
On parie que vous avez déjà écrit ?

- Tests de non régression
- Benchmarking de code
- Profilage de code (image)
- Optimisation de code
- Génération de fichier PDF & ODF
- Perl 6 : éléments du langage
- Ouverture à la communauté



Line	Host	Type	Size	Time	Code
1	192.168.1.1	HTTP	1024	0.0000	200 OK
2	192.168.1.1	HTTP	1024	0.0000	200 OK
3	192.168.1.1	HTTP	1024	0.0000	200 OK
4	192.168.1.1	HTTP	1024	0.0000	200 OK
5	192.168.1.1	HTTP	1024	0.0000	200 OK
6	192.168.1.1	HTTP	1024	0.0000	200 OK
7	192.168.1.1	HTTP	1024	0.0000	200 OK
8	192.168.1.1	HTTP	1024	0.0000	200 OK
9	192.168.1.1	HTTP	1024	0.0000	200 OK
10	192.168.1.1	HTTP	1024	0.0000	200 OK

Option *Perl*

Comment travailler cette option ?

- En relisant les cours :
 - faire un index des notions abordées,
 - prendre des notes (dégager les points importants),
 - faire une fiche synthétique (format A5?) par cours.
- En parcourant les notions déjà vues avant de venir en cours.
- En refaisant les TD (sans le cours ?) : sur machine, sur papier.

Évaluation de l'option : examen de 2 heures sur papier
 (tous documents papier autorisés, ordinateurs/téléphones interdits)

Scalaire, tableaux, fonctions



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

La reproduction et/ou diffusion de ce document, même partielle, quel que soit le support, numérique ou non, est strictement interdite sans autorisation écrite des ayants droit.



Plan des 4 premiers cours Perl

Introduction	Tableaux de hachage
Prise en main	Expressions régulières
Scalaire	Fichiers, entrée/sorties
Structures de contrôle	Références
Listes et tableaux	Modules
Fonctions et programme	Programmation objet
Fonctions sur les listes	



Introduction

Practical Extraction and Report Language

Créé en 1987 par Larry Wall.

Stable : 5.32.0 / 5.30.3

Développement : 5.33

Une mineure par an. Mai/juin 2021 : 5.34 / 7?



L'oignon, le symbole de Perl

Inspiration : C, shells, sed, grep, awk, lisp

Langage interprété pré-compilé à l'exécution.

Licence de l'interpréteur : GNU GPL / *Artistic License*

Perl : le langage

perl : l'interpréteur

There is more than one way to do it.



Versions de Perl

Branche	Date	État	À jour	Date
5.34 / 7	2021-05-??	Développ.	5.33.x	
5.32	2020-06-20	Maintenue	5.32.0	2020-06-20
5.30	2019-05-22	Maintenue	5.30.3	2020-06-01
5.28	2018-06-23	Fin de vie	5.28.3	2020-06-01
5.26	2017-05-30	Fin de vie	5.26.3	2018-11-29
5.24	2016-05-09	Fin de vie	5.24.4	2018-04-14
5.22	2015-06-01	Fin de vie	5.22.4	2017-07-15
5.20	2014-05-27	Fin de vie	5.20.3	2015-09-12
5.18	2013-05-18	Fin de vie	5.18.4	2014-10-02
5.16	2012-05-20	Fin de vie	5.16.3	2013-03-11
5.14	2011-05-14	Fin de vie	5.14.4	2013-03-10
5.12	2010-04-12	Fin de vie	5.12.5	2012-11-10
5.10	2007-12-18	Fin de vie	5.10.1	2009-08-23



Domaines de prédilection

Ce qui fait la différence entre langages, ce n'est pas ce qu'ils rendent possible, mais ce qu'il rendent facile.

- Prédilection pour les données textuelles (à la mode!) :
 - Bases de données
 - Flux et protocoles réseaux : web, courriel, SNMP, LDAP, etc
 - Administration système : logs, configuration
 - Manipulation de formats de données : XML, CSV, etc
 - Algo du texte : génomique, linguistique
- Passerelles applicatives / inter-SI / framework
- Autre : graphisme, système, archivage, chiffrement, GUI, etc

Couteau suisse du monde Linux/Unix

Aide à l'interopérabilité : **glue générale entre presque tout**



Avantages

- Un vrai langage puissant :
 - Paradigmes : impératif / fonctionnel / orienté objet,
 - Récursivité / modularité / exceptions / ramasse-miettes,
 - Tableaux, listes et tables de hachage natifs,
 - Expressions régulières, support natif d'unicode,
 - Surcharges d'opérateurs, fermetures (closures),
- Richesse des bibliothèques (efficacité de programmation),
- Ouverture d'esprit de la communauté Perl
Perl beaucoup utilisé par admins, culture d'outils multiples
⇒ large culture de l'informatique, tolérance
- *There is more than one way to do it.*



Inconvénients

- Langage faiblement typé (scalaires),
- Langage très permissif, liberté de coder
 - utilisateurs hétérogènes, existant peu recommandable parfois,
 - exige rigueur et coordination (liberté de choisir ses contraintes),
- Riche donc pas immédiat à maîtriser
- Faible protection attributs et méthodes (nécessité sociale non technique),
- Ramasse-miettes par comptage de références (données cycliques)
- Ce n'est pas à la mode ;-)
Ce n'est jamais la bonne solution technique qui est adulée :
8086 vs 68000, Windows vs Linux, The Voice vs chanteurs :-)
- *There is more than one way to do it.*



Sources d'information web sur Perl

- CPAN : distributions et modules
<https://www.cpan.org/> <https://metacpan.org/>
- perldelta : changements dans le langage à chaque version
<https://metacpan.org/pod/distribution/perl/pod/perldelta.pod>
- Pour démarrer : pierres de Rosette
<http://hyperpolyglot.org> principaux langages
<http://rosettacode.org> 515 langages
- Guide Perl (S. Lhullier)
<https://formation-perl.fr/guide-perl.html>
- Association *Les Mongueurs de Perl*
<http://mongueurs.net/>



Sources d'information : perldoc

`perldoc perl` – index
`perldoc perlop` – opérateurs du langage
`perldoc perlfunc` – fonctions du langage
`perldoc perlfaq` – FAQ
`perldoc perlre` – regexp
`perldoc perlrun` – options de la ligne de commande
etc

`perldoc -f fonction` `perldoc -f sprintf`
`perldoc -q mots-clefs-FAQ` `perldoc -q 'macros for vi'`
`perldoc module` `perldoc Net::FTP`



Bibliographie

Learning Perl / Introduction à Perl
Schwartz - O'Reilly



Le «camel book» ...

Programming Perl / Programmation en Perl
Wall, Christiansen & Orwant - O'Reilly

Perl best practices / De l'art de programmer en Perl
Conway - O'Reilly

Donne à quelqu'un du poisson, il mangera un jour. Donne lui une canne à pêche, il mangera tous les jours. Donne lui un bouquin de référence avec un gros index, il mourra de faim en passant son temps à le consulter.
Rafael Garcia-Suarez



Bibliographie

Perl cookbook / Perl en action
Christiansen & Torkington - O'Reilly

Advanced Perl programming / Programmation avancée en Perl
Srinivasan - O'Reilly

Object Oriented Perl
Damian Conway - Manning Publications Company

Mastering algorithms with Perl
Orwant, Hietaniemi & MacDonald - O'Reilly

Perl testing : a developer's handbook
Langworth & Chromatic - O'Reilly

Perl moderne
Krotkine, Aperghis-Tramoni, Quelin, Bruhat - Pearson



Quick reference guides

<http://www.ch.embnet.org/CoursEMBnet/Pages05/slides/perlref.pdf>

https://michaelgoerz.net/refcards/perl_refcard.pdf

http://www.cheat-sheets.org/saved-copy/perl_refcard.pdf

<http://johnbokma.com/perl/perl-quick-reference-card.pdf>

<http://www.rexswain.com/perl5.html>

Regexp :

<http://people.duke.edu/~tkb13/courses/ece560/resources/preqr.pdf>

32 pages Perl 5 :

<http://www.squirrel.nl/pub/perlref-5.004.1.pdf>

<http://www.netzmafia.de/skripten/perl/perl-qref.pdf>

Et le vôtre !



Prise en main

- `perl -v` – affiche le numéro de version
- `perl -V` – affiche les options de compilation de l'exécutable perl, les répertoires des bibliothèques Perl (tableau `@INC`) ...
- `perl -d` – lance le débogueur
- `perl -de 1` – mode interactif



Lancer un programme Perl

- Code en ligne de commande

```
perl -e 'print("Salut Larry !\n");'
```

Salut Larry !
- Script en argument
fichier `salut.pl`

```
print("Salut Larry !\n");
```

```
perl salut.pl
```

Salut Larry !



Lancer un programme Perl

- Shebang (LA bonne méthode)
fichier `salut.pl`

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

```
print("Salut Larry !\n");
```

```
chmod +x salut.pl
```

```
./salut.pl
```

Salut Larry !
- Shebang pour Perl non standard

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

```
print("Salut Larry !\n");
```



Pragmas

- `use strict;`
Modifie la phase de compilation. Langage moins permissif : obligation de déclaration des variables, interdiction des références symboliques, vérification de l'existence des fonctions appelées ...
- `use warnings;` (anciennement flag `-w`)
Modifie la phase d'exécution, affichage d'avertissements : variable non initialisée, conversion numérique partielle ...
- `use diagnostics '-l=fr';`
Indique à Perl de produire les messages dans leur forme longue
- `use 5.010;`
Prise en compte des fonctionnalités expérimentales depuis 5.10



Structures du langage

Point-virgule à la fin de chaque instruction : `print($v);`

Espacements libres :

- `if($v==$w){print"Ok\n";}`
- `if ($v == $w) {
 print "Ok\n";
}`

Commentaires : depuis un `#` jusqu'à la fin de la ligne.

```
# Ceci est un commentaire  
print("$s\n"); # Et cela aussi
```

Toujours coder comme si le gars qui va finir par maintenir votre code est un psychopathe violent qui sait où vous habitez. – Rick Osborne



Types de données

Simplicité, souplesse et puissance.

- Introspection,
- Structures anonymes,
- Autovivification,
- Support natif d'unicode.

Trois types :

- Scalaire : donnée atomique,
- Tableau (gestion dynamique et automatique de la taille),
- Table de hachage (association clef → valeur) *NB: performance.*



Structures de donnée

Sigil :

`$` scalaire : `$x=42; $y=31.8; $z='bonjour';`

`@` tableau de scalaire : `@t = (3,'vf',-84)`

`%` table de hachage :

`%h = ('paul'=>'arg','pierre'=>-84.2,'julie'=>'aussi')`

structure	globalité	élément
Scalaire	<code>\$v</code>	
Tableau	<code>@v</code>	<code>\$v[indice]</code>
Hash	<code>%v</code>	<code>\$v{clef}</code>



Structures de donnée : scalaires

Donnée atomique : chaîne de caractères, nombre ...

12 -3.14 "chaîne" 'chaîne' 3e9 034 0xFF

Variable : `$x`

- déclaration :

`my $x;` `my $x = 10;`

Portée (visibilité) : le bloc et ses sous-blocs.

Place libre dans le bloc.

- affectation et utilisation :

```
$x = $y + 3;  
$phrase = "Bonjour $prenom !";  
print("$phrase\n");
```



Délimiteurs de chaînes

- double quote : *interprétation* du contenu :
 - `"Salut\n"` une chaîne + new-line
 - `"Salut $prenom\n"` *substitution/interpolation* des variables
 - `"\t"` une tabulation
 - `"\" " \" " \"\$ " \"@"` *protection, déspecialisation*
- simple quote : pas d'*interprétation* du contenu :
 - `'Bonjour'` une chaîne
 - `'Bonjour\n'` le `\n` n'a pas de sens ici
 - `'Bonjour $prenom'` pas de *substitution* des variables
 - `'\"' ' \"' ' \"\$' ' \"@'` *protection, déspecialisation*
- opérateurs `q()` `q<>` `q!!` `qq()` `qq<>` `qq!!`



Délimiteurs de nom de variable

Dans les doubles quotes, les variables sont interpolées :

`"Salut $prenom\n"`

Comment concaténer une variable à une chaîne ?

Exemple : verbe `-port-` et préfixe `ap`, `sup`, `im`, `ex`, `dé`, `com`, etc

`"il $prefixeport$suffixe"`

⇒ Variable `$prefixeport`

Possibilité de sortir la variable des doubles quotes :

`"il ".$prefixe."port$suffixe"`

Mais faisable dans les doubles quotes :

`"il ${prefixe}port$suffixe"`

Délimiteur de ce sur quoi le sigil s'applique : `{ }`

⇒ Utile plus tard



Afficher

- `print "$v\n", "$n\n";`
Affichage des scalaires passés en paramètres
- `printf "%s\n%d\n", $v, $n;`
Affichage selon un code de format des scalaires passés en paramètres
- `use 5.010; # À placer en haut de script`
`say $v;`
Affichage des scalaires passés en paramètres suivis d'un retour à la ligne (depuis Perl 5.10)



La valeur undef

`undef` est une valeur particulière scalaire

- signifie «non défini»
- valeur par défaut des variables :
`my $x; ⇔ my $x=undef;`
- vaut 0 en contexte numérique, `''` en contexte de chaîne
- affectation : `$x = undef;` ou `undef($x);`
- test : `if(defined($x))`
- Ne pas écrire : `if($x != undef)` ou autre
- Opérateur `//` `//=`



Opérateurs

- Arithmétiques : `+` `-` `*` `/` `%` `**` puissance
- Chaînes :
 - `.` concaténation `$x='bonjour' '.$prenom;`
 - `x` répétition `$x='bon'*3 ⇒ 'bonbonbon'`
- Fonction `length`
`length($x)` renvoie le nombre de caractères de la chaîne `$x`
`length('bonjour')` vaut 7
- Fonctions de manipulation de la «case» :
 - `lc` renvoie la chaîne mise en minuscule
 - `lcfirst` renvoie la chaîne avec la 1ère lettre en minuscule
 - `uc` renvoie la chaîne mise en majuscule
 - `ucfirst` renvoie la chaîne avec la 1ère lettre en majuscule`$nom = ucfirst(lc($nom));`



Raccourcis

- Raccourcis
 - `$v = $v + 4;` \Rightarrow `$v += 4;`
 - `$v = $v.'mot';` \Rightarrow `$v .= 'mot';`
 - `-=` `*=` `/=` `%=` `x=` `**=`
- {In,Dé}crémenteurs
 - `$v = $v + 1;` \Rightarrow `$v++;` ou `++$v;`
 - `$v = $v - 1;` \Rightarrow `$v--;` ou `--$v;`
 - `$v++` `++$v` même effet de bord
 - `$v++` valeur de `$v` avant incrémentation
 - `++$v` valeur de `$v` après incrémentation



Fonction substr

`substr($x, offset, length)`
 vaut la sous-chaîne de position *offset* et de longueur *length*

- `substr('bonjour',2,3)` vaut `'njo'`
- *length* peut être omis : toute la partie droite est sélectionnée.
- 4ème paramètre : valeur de remplacement
`my $y = 'salut toi';`
`substr($y, 5, 1, 'ation à ');`
 \Rightarrow `$y` vaut `'salutation à toi'`
 \Rightarrow Perl gère lui-même la mémoire !



Typage dynamique

Les scalaires ne sont pas typés, mais les valeurs stockées le sont :

- `$x = '1';` \Rightarrow stockage en chaîne (suite de caractères)
- `$x = 1;` \Rightarrow stockage en entier (4 octets, complément à 2)
- `$x = 1.0;` \Rightarrow stockage en flottant (8 octets, norme IEEE 754)

Influence sur la performance (calculs).



Notion de contexte

Valeur d'expression : dépend de la position dans le code (le contexte)

Types de contexte :

- scalaire (numérique, chaîne, booléen) : monovalué
- de liste (lire la suite) : multivalué

Par exemple, où trouve-t-on les contextes ?

- booléen ? Dans les tests : `if($x) { ... }`
- de chaîne ? Dans les concaténations : `$x . $y`
- numérique ? Dans les calculs : `$x + $y`

Règles de conversion chaîne→numérique :

- `'23.5'` → 23.5
- `'23.5re'` → 23.5 ⇒ message d'avertissement si `use warnings;`
- `'re23.5'` → 0 ⇒ message d'avertissement si `use warnings;`



Constantes

```
use constant DEBUG => 0;
use constant {
    PROTO => 'http',
    PORT => 80,
};
print PROTO."\n" if DEBUG;

use constant PI      => 4*atan2(1,1);
# Calcule la valeur une bonne fois pour toute

use constant JOURS => qw(lundi mardi mercredi
                          jeudi vendredi samedi dimanche);
print 'Premier jour de la semaine : ', (JOURS)[0], ".\n";
```



Structures de contrôle : conditions

- `if(condition){ code1 } (else { code2 })`

NB: { } obligatoires

```
if( $x == 0 ) {
    print "$x nul\n";
} else {
    print "$x non nul\n";
}
```

Incorrect :

```
if( condition1 ) { code1 }
else if( condition2 ) { code2 }
```

Correct :

```
if( condition1 ) { code1 }
else { if( condition2 ) { code2 } }
```



Structures de contrôle : conditions

- `elsif` pour ne pas surcharger en accolades :

```
if( condition1 ) { code1 }
elsif( condition2 ) { code2 }
elsif( condition3 ) { code3 }
else { code4 }
```

```
if( $x == 0 ) {
    print "$x nul\n";
} elsif( $x < 0 ) {
    print "$x négatif\n";
} else {
    print "$x positif\n";
}
```



Structures de contrôle : conditions

- *instruction if condition ;*

```
print "$s\n" if defined $s;
```

 Pas de parenthèse obligatoire, ni d'accolade.
 Pas de `else` possible.
- `unless(condition) ⇔ if(!condition)`



Booléens et opérateurs logiques

Booléens : scalaires

- Valeurs fausses : `'', undef, 0, '0'`
- Valeurs vraies : `1, 'blabla', -4.2, '00', 0.0`, etc
- `if($x) { ... }`

Opérateur logiques, 2 familles :

- Forte priorité `! && ||`
- Faible priorité `not and or`

`if((a || b) && (c || d))`

`if(a || b and c || d)`

`perldoc perlop`



Opérateurs de tests

Les opérateur de tests imposent un contexte à leur opérandes.

contexte imposé	numérique	chaîne
égalité	<code>==</code>	<code>eq</code>
différence	<code>!=</code>	<code>ne</code>
infériorité	<code><</code>	<code>lt</code>
supériorité	<code>></code>	<code>gt</code>
inf ou égal	<code><=</code>	<code>le</code>
sup ou égal	<code>>=</code>	<code>ge</code>
comparaison	<code><=></code>	<code>cmp</code>

```
if( '0' == '00' )           if( $x != undef )
if( '0' eq '00' )           ⇒ if( defined $x )
```



Structures de contrôle : boucles

- `for(initialisation; condition; incrément) { code }`

```
for( my $i = 0;
      $i <= 10;
      $i++ ) {
  print "$i\n";
}
```

```
for( my $v=100; $v>=0; $v-=2 ) {
  print "$v\n";
}
```



Structures de contrôle : boucles

- `while(condition) { code }`
instruction while condition;
`do { code } while(condition);`

```
my $e = 0;
while( $e <= 10 ) {
    print "$e\n";
    $e++;
}
```

- `until(condition) ⇔ while(!condition)`



Structures de contrôle : boucles

Sauf pour `do while` :

- `last` : sortie de boucle
- `next` : arrêt de l'exécution du bloc
 (retour à l'évaluation de la condition)
- `redo` : recommence l'exécution du bloc courant
 (sans passer par l'évaluation de la condition)

```
for( my $e=0 ; $e<=10; $e++ ) {
    next if $e % 2 == 0; # ne pas traiter les pairs
    print "$e\n";
    last if $e == $valeur; # mettre fin à la boucle
}
```



Premier exemple simple

```
#!/usr/bin/perl
use strict;
use warnings;
my $somme = 0;
for( my $i=9; $i>0; $i-- ) {
    if( $i % 2 != 0 ) {
        print("$i impair\n");
    }
    print( '- 'x$i . "\n" );
    if $i % 3 == 0;
    $somme += $i;
}
print("$somme\n");
```

```
9 impair
-----
7 impair
-----
5 impair
3 impair
---
1 impair
45
```



Type de données : liste

- liste vide : `()`
- listes : `(2,5,1,0,-4)` ou `(2,'âge','bonjour $prenom')`
 \Rightarrow liste \equiv liste de scalaires
- 256 éléments : `(0..255)`
- 37 éléments : `(1..10,'âge','a'..'z')`
- `($debut..$fin)` *pas toujours possible* \Rightarrow liste vide
- `(1,2,(1..10),'âge','b',6),4)` \Rightarrow une liste à 16 éléments
 «Aplatissage» ou «linéarisation» des listes
- `(2,3)x4` \Rightarrow `(2,3,2,3,2,3,2,3)`



Structure de données : tableau

Tableau \equiv mono-dimensionnel

- déclaration d'un tableau : `my @t;`
`my @t = ();`
 NB: Pas de taille à déclarer (taille dynamique)
- déclaration et initialisation d'un tableau par une liste :

```
my @t = (23, 'ab', -54.4);
```

$$\begin{array}{c} @t \\ \begin{array}{|c|c|c|} \hline 0 & -3 & 1 \\ \hline 23 & ab & -54.4 \\ \hline \end{array} \end{array}$$

```
my @t = (4..12); # 9 valeurs
```

- accéder à un élément : `$t[i]` Attention : `$`
`$t[0]` : premier élément, puis `$t[1]` `$t[2]` etc
`$t[-1]` : dernier élément, puis `$t[-2]` `$t[-3]` etc



Tableau : éléments

- Éléments non définis : `undef`
`$t[3]` et suivants, `$t[-4]` et précédents)

$$\begin{array}{c} @t \\ \begin{array}{|c|c|c|} \hline 0 & -3 & 1 \\ \hline 23 & ab & -54.4 \\ \hline \end{array} \end{array}$$

- Ajouter un élément au tableau :
`my @t = (23, 'ab', -54.4);`
`$t[3] = 'def';` (impossible avec les indices négatifs)
 \Rightarrow Allongement du tableau
 \Rightarrow Perl gère lui-même la mémoire



Tableau

- `$t[100]=6;` \Rightarrow les éléments intermédiaires n'existent pas

$$\begin{array}{c} @t \\ \begin{array}{|c|c|c|} \hline 0 & -101 & 1 \\ \hline 23 & ab & -54.4 \\ \hline \end{array} \dots \begin{array}{|c|} \hline 100 \\ \hline 6 \\ \hline \end{array} \end{array}$$

Attention : tableau à trous, à éviter !

- On peut tester l'existence d'un élément :
`if (exists($t[1])) { ... }`

- `$#t` : indice du dernier élément du tableau `@t` (affectable!)
`my @t = ('a'..'z');`
`$#t = 12;`
 \Rightarrow le tableau réduit de taille (perte des valeurs)



Tableau : substitution et contexte

- Substitution entre doubles-quotes :
 \Rightarrow représentation des éléments séparés par un espace.

```
my @t = ( -12.5, 'he', 0xFF );
```

```
my $s = "@t";  $\Rightarrow$  '-12.5 he 255' print "@t\n";
```

- En contexte scalaire (chaîne, numérique, booléen) :
 \Rightarrow renvoie sa taille.

```
my $v = @t;  $\Rightarrow$  3
```

```
my $v = scalar( @t );
```

```
@t."
```

```
@t+10
```

```
if( @t ) { ... }
```

```
if( @t > 0 ) { ... }
```

```
print(@t."
```

```
print("@t\n");
```

**Liste et tableau : affectations et contexte**

- Affectation et liste :


```
($x,$y) = (1,2);
($x,$y) = (1);
($x,$y) = (1,2,3);
($x,$y) = Fonction();
($x,$y) = 1;    ⇒ Comme le cas ($x,$y) = (1);
($x,$y) = ($y,$x);
```

Perl ne se prend pas les pieds dans le tapis : très utile plus tard
- Affectation et tableau :


```
($x,$y) = @t; # Contexte de liste
@t = ($x,$y); ⇒ Perte des anciennes valeurs de @t
@t = 1;       ⇒ Comme le cas @t = (1);
```
- Affectations différentes : `$x=@t;` `($x)=@t;`

**Liste/tableau : affectation et déclaration multiples**

- Différence d'affectation depuis liste et tableau :


```
$x = @t;           ⇒ taille du tableau
$x = ($y,$z);      ⇒ copie de $y
$x = Fonction();   ⇒ copie du 1er élément renvoyé
```
- Déclaration multiple :


```
my $x,$y; ⇒ erreur
my ($x,$y);
my ($x,$y) = (1,2);
```
- Copie de tableau à tableau : `@t2 = @t;`

**Liste : «aplatissement»**

- Right-value :


```
@t = (1,2,'age');
@t2 = (10,@t,20);
⇒ @t2 vaut la liste (10,1,2,'age',20)

print(@t,"\n"); # Aplatissement
print(@t."\n"); # Contexte scalaire
print("@t\n");  # Interpolation
```
- Left-value :


```
($x,@t) = @s;
  $x vaut le premier élément de @s
  @t «absorbe» tous les autres éléments
($x,@t,$y,@t2) = @s;
```

**Manipulation de tableaux et listes**

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

- Ajout et suppression à gauche dans un tableau :


```
- unshift(@t,5,6);    ⇒ @t=(5,6,1,2,3,4)
- $v = shift(@t);     ⇒ $v=1 @t=(2,3,4)
```
 - Ajout et suppression à droite dans un tableau :


```
- push(@t,5,6);       ⇒ @t=(1,2,3,4,5,6)
- $v = pop(@t);       ⇒ $v=4 @t=(1,2,3)
```
 - Inversion de liste : `@s = reverse(@t);` `⇒ @s=(4,3,2,1)`
- Comment remplacer ces fonctions par des instructions du langage ?



Tableau : splice et @ARGV

Remplacer des éléments : `splice(tableau,début,nbr,remplacement)`

Si pas de remplacement, suppression :

```
my @t = ('a'..'z');
@s = splice( @t, 3, 2 ); # Supprime et renvoie ('d','e')
```

Remplacer les éléments supprimés :

```
@s = splice( @t, 3, 2, ('A'..'Z') );
```

@ARGV : tableau des arguments de la ligne de commande
(nom du programme : \$0)



Structures de contrôle : foreach

- `foreach variable (liste) { code }` parcours non destructif
La variable est un alias pour chaque **valeur** successive de la liste.
NB1: Ne pas modifier la liste lors parcours
NB2: Modifier le variable modifie la liste
- `foreach $v (1,2,3,4) { ... }`
`foreach $v (@t) { ... }`
`foreach $v (@t,5,@t2,$x,'azerty') { ... }`
`foreach $v (1..10) { ... }`
`foreach $v (reverse @t) { ... }`
- `foreach my $v (liste) { code }`
`foreach (@t) { fonction($-); }` *\$- variable par défaut*
- `next` valeur suivante (fin de l'exécution du bloc)
`last` sortie de boucle



Création de listes avec l'opérateur qw

- Il peut être pénible d'écrire :
`@t = ('Ceci', 'est', 'un', 'peu', 'ennuyeux,', 'non', '?');`
- L'opérateur `qw` découpe une chaîne et renvoie une liste
Séparateur : les caractères espaces ou `\n\t\r\f (\s+)`
`@t = qw(Cela est bien plus facile, non ?);`
- `@t = qw/ attention 'aux erreurs' stupides /;`
⇒ 4 éléments
- `foreach $v (qw(liste de mots)) { ... }`



Concept de fonction

Regroupement d'instruction qui peut être utilisé plusieurs fois.

Prend des valeurs en entrée (appelées paramètres)
et renvoie des valeurs de retour.

Comporte des instructions et des appels à d'autres fonctions.

Deux phases : *une* déclaration puis *des* utilisations

Une même fonction peut être appelée avec des paramètres différents :

```
my $val1 = calcul( 2, 'aa' );
my $val2 = calcul( 10, 'bb' );
```

`print push length` sont des fonctions.

`if for foreach qw` ne sont pas des fonctions mais des instructions.



Écrire une fonction

- `sub calcul {`
`my ($nom,$age,$nbr) = @_; #arguments scalaires`
`...`
`return $r;`
`}`
`my $val1 = calcul('Jean',42,7.5);`
`my $val2 = calcul('Paul',18,-0.4);`
- Si 1 seul argument : `my ($x) = @_;`
→ () obligatoires
- `my ($x,@t) = @_;`
→ @t reçoit les arguments restant



Écrire une fonction

- Appel :
`calcul('Jean',42,7.5);` `$v = calcul('Jean',42);`
`$v = calcul('Jean',42,7.5);` `$v = calcul('Jean',42,7.5,'Paul');`
`calcul 'Jean',42,7.5;` (à éviter)
- Renvoyer une liste :
`return ($y,$z);`
`return @t;`
Appel :
`($j,$k) = calcul('Jean',42,7.5);`
`@s = calcul('Jean',42,7.5);`



Appel de fonction

Utiliser `calcul('Jean',42,7.5);` ou `calcul 'Jean',42,7.5; ?`

Plus léger sans parenthèses, pour les fonctions de bases :
`print pop map etc`

Cas à problème :

```
print ($i*$j)."\n";
⇒ la priorité n'est pas celle qu'on croit
print ( ($i*$j)."\n");
```



Tableaux et appel de fonction

- Appel : par copie (aplatissement)
`action(@t, @t2); # Aplatissement !`

```
sub action {
    my (@v,@w) = @_; # @w est vide
    [.....]
}
```
- Retour : par copie (aplatissement)

```
sub action {
    [.....]
    return @t, @t2; # Aplatissement !
}
(@v,@w) = action(); # @w est vide
```




Prototype de fonction (expérimental)

```
use v5.20;
use feature qw(signatures);
no warnings qw(experimental::signatures);
sub f1( $x, @t ) { # également @_
    print "$x\n";
    print "@t\n";
}
f1( 'texte', 1..10 );
```

Ne règle pas le problème de l'aplatissement :

```
sub f2( @t, @t2 ) { ... }
```

Shurpy parameter not last at fichier.pl line 12.



Structure d'un programme

<code>#!/usr/bin/perl</code>	<i>shebang</i>
<code>use strict;</code>	<i>règles strictes</i>
<code>use warnings;</code>	<i>activation des avertissements</i>
<code>my \$t = 'Bonjour Larry';</code>	<i>variable globale</i>
<code>print "\$t\n";</code>	<i>avec ou sans ()</i>
<code>sub SommeProduit {</code>	
<code>my (\$x,\$y) = @_;</code>	<i>paramètres de la fonction</i>
<code>my \$m = \$x*\$y;</code>	<i>variable locale</i>
<code>printf("%5d\n", \$m);</code>	<i>printf comme en C</i>
<code>return (\$x+\$y,\$m);</code>	<i>valeurs de retour de la fonction</i>
<code>}</code>	
<code>my @t = SommeProduit(3,5);</code>	<i>NB: pas de conflit entre \$t et @t</i>
<code>print "\$t[0] \$t[1]\n";</code>	<i>ni avec %t ou sub t</i>
<code>my @t2 = SommeProduit(2,7);</code>	



Liste : joindre et découper en chaîne

- `join` : joindre les éléments dans une chaîne

```
scalaire = join( séparateur, liste );
$s = join(':', @t);
$v = join(',', $x,$y,$z);
```
- `split` : découper une chaîne en liste

```
liste = split( /séparateur/, chaîne );
@t = split(/,/, $v);
($x,$y) = split(/:/, $v);
```
- `print join(':', split(/!/, 'salut!ici'));`
- `foreach my $mot (split / /, $texte) { ... $mot ... }`



Liste : effectuer une sélection

- `grep` : sélectionner les éléments

```
liste2 = grep { test } liste1 ;
liste2 = grep /regex/, liste1 ;
```

→ sélection des éléments pour lesquels l'expression est vraie.
 \$_ vaut localement chaque élément de liste1

```
@s = grep { $_ > 4 } @t;
@s = grep { length($_) > 10 } @t;
@s = grep { $_ % 2 != 0 } @t;
@s = grep { $_>0 and $_!=4 } @t;
@s = grep { monTest($_) } @t;
@s = grep /^aa/, @t;
```



Liste : appliquer un traitement

- **map** : appliquer une fonction à tous les éléments
`liste2 = map { expression } liste1 ;`
 → création de la liste où chaque élément vaudra
expression de chaque élément de *liste1*
`$_` vaut localement chaque élément de *liste1*

```
@s = map { $_ * 4 } @t;
@s = map { $_ * $_ } @t;
@s = map { uc($_) } @t;
@s = map { ' '. $_ . ' ' } @t;
@s = map { monCalcul($_) } @t;
```

Modifier `$_` modifie *liste1* : `map { $_ *= 4 } @t;`



Opérateurs de comparaison

`<=>` comparaison numérique (contexte numérique) *spaceship*
`cmp` comparaison alphabétique (contexte de chaîne) «*ASCIIbétique*»

`$x<=>$y` vaut :

- un nombre positif, si `$x>$y` en tant que nombres,
- un nombre négatif, si `$x<$y` en tant que nombres,
- zéro, si `$x` vaut `$y` en tant que nombres.

`$x cmp $y` idem selon l'ordre alphabétique («*ASCIIbétique*»).

⇒ Opérateurs utilisés pour trier

Comparer	11	aa
	2	b



Liste : trier

- **sort** : trier une liste
`liste2 = sort liste1 ;`
`liste2 = sort { comparaison } liste1 ;`
`$a` et `$b` valent localement chaque couple (*source d'erreurs!*)
`@s = sort { $a cmp $b } @t;` tri lexical (défaut)
`@s = sort { $a <=> $b } @t;` tri numérique
`@s = sort { length($a) <=> length($b) } @t;`
`@s = sort { substr($a,0,5) cmp substr($b,0,5) } @t;`
 (le tri est dit *conservatif*)
`@s = sort { maComparaison($a,$b) } @t;`
`maComparaison` doit renvoyer un nombre :
 positif si `$b` doit être avant `$a`,
 négatif si `$a` doit être avant `$b`,
 nul s'ils sont équivalents



Fonctions sur les listes

Ces fonctions sont auto-applicables :

- `@t = grep { $_ >= 10 } @t;`
- `@t = map { $_ * $_ } @t;`
- `@t = sort { $a <=> $b } @t;`

Chainage possible entre fonctions :

```
foreach my $e ( sort { $a <=> $b }
                map { $_ * $_ }
                grep { $_ >= 10 }
                @t )
{ ... }
```