

Introduction au langage Perl

Bruno Pinaud

Site de référence : <http://www.perl.org>

Introduction

- PERL : Practical Extraction and Reporting Language
- Langage générique de haut niveau interprété et dynamique initialement développé pour manipuler des chaînes de caractères.
- Mis au point par Larry Wall en 1987 (toujours actif sur le développement de Perl).
- Langage très flexible : TMTOWTDI
 - There is More Than One Way To Do It.
- Langage très riche : nombreux modules additionnels disponibles : <http://cpan.perl.org>
- Voir aussi sur votre ordinateur : `perldoc -f <mot clés>`

Introduction

- Objectifs principaux de ce cours : avoir une démarche scientifique de traitement de données.
- A partir de données brutes dans un format quelconque
 - Compréhension et nettoyage des données
 - Mise en forme dans un format maîtrisé
 - Interrogation des données
 - Restitution en utilisant notamment différents tableaux de bords

Introduction : avantages de Perl

- Langage **interprété**, **impératif** et faiblement **objet**,
- Nombreuses propriétés de haut niveau :
 - **gestion automatique de la mémoire**,
 - **typage dynamique**
 - présence de **nombreux types fondamentaux** (strings, listes, table de hachage, . . .)
 - **expressions régulières**,
 - introspection et fonction d'évaluation, c'est à dire réflexivité, introspection et intercession.

Introduction : helloWorld.pl

```
# !/usr/bin/perl  
#mon premier programme Perl  
print "Hello World!\n" ;
```

Exécution :

```
~$ perl helloWorld.pl
```

Ou

```
~$ chmod +x helloWorld.pl
```

```
~$ ./helloWorld.pl
```

Les types 1/2

- Un **scalaire** (scalar) est une donnée simple, i.e. un nombre, une chaîne de caractères ou une référence.
- Exemples
 - `$foo=42 ; #un scalaire`
 - `$bar= "abc" ; #un autre scalaire`

Les types 1/2

- Un **scalaire** (scalar) est une donnée simple, i.e. un nombre, une chaîne de caractères ou une référence.
- Une **liste** (array) est un ensemble ordonné de scalaires.
- Exemples
 - `$foo = 42 ; # un scalaire`
 - `$bar = "abc" ; # un autre scalaire`
 - `@foo = (1,2,3,4) ; # une liste`
 - `print "$foo[2] \n" ; # accès à un élément de la liste => 3`
 - `print "$#foo \n" ; # indice du dernier élément de la liste`

Les types 1/2

- Un **scalaire** (scalar) est une donnée simple, i.e. un nombre, une chaîne de caractères ou une référence.
- Une **liste** (array) est un ensemble ordonné de scalaires.

Exemples

- `$foo = 42 ; # un scalaire`
 - `$bar = "abc" ; # un autre scalaire`

 - `@foo = (1,2,3,4) ; # une liste`
 - `print "$foo[2] \n" ; # accès à un élément de la liste => 3`
 - `print "$#foo \n" ; # indice du dernier élément de la liste`
-
- L'interprétation d'une variable dépend aussi du contexte, exemple :
- `print $foo+2 ; # la longueur de la liste+2`

Les types 2/2

- Une **table de hachage** (hash) est un ensemble de paires de scalaires (une clé pour l'index et une valeur associée),
- Exemples
 - `%t=('1',4,'abc',"gfr"); #un hash`
 - `print "${t['1']}\n"; #accès à la valeur associée à la clé '1'`
 - `print "${t['abc']}\n"; #accès à la valeur associée à la clé 'abc'`

Les types 2/2

- Une **table de hachage** (hash) est un ensemble de paires de scalaires (une clé pour l'index et une valeur associée),
- Un **descripteur de fichier** (file handle) est un lien vers un fichier, un périphérique ou un pipeline ouvert

- Exemples
 - `%t=('1',4,'abc',"gfr"); #un hash`
 - `print "${'1'}\n"; #accès à la valeur associée à la clé '1'`
 - `print "${'abc'}\n"; #accès à la valeur associée à la clé 'abc'`
 - `print STDERR "Error !\n"`

Les structures de contrôles

- Les blocs d'instructions sont toujours entourés d'accolades {...}.

Les structures de contrôles

- Les blocs d'instructions sont toujours entourés d'accolades {...}.
- Les **conditions** :
- Exemples
 - `If (expr) { bloc d'instructions }`
`elseif (expr) { bloc d'instructions }`
`else {bloc d'instructions}`
 - `If ($a==5 and $mot eq "oui") {`
`print "ok!\n"`
`} else {`
`print "raté!\n"`
`}`

Les structures de contrôles

- Les blocs d'instructions sont toujours entourés d'accolades {...}.
- Les **conditions** :
- Exemples
 - `if(expr) { bloc d'instructions }`
`elseif(expr) { bloc d'instructions }`
`else { bloc d'instructions }`
 - `if($a==5 and $mot eq "oui") {`
`print "ok!\n"`
`} else {`
`print "raté!\n"`
`}`
 - `print "Hello !" if ($t eq "ttt");`
 - `unless (expr) { bloc d'instructions }` `# if inversé`

Les structures de contrôles

- Les **boucles** :
- Exemples
 - `for (init-expr ; cond-expr ; incr-expr) { bloc d'instructions }`
 - `for $x (liste) { bloc d'instructions }`

Les structures de contrôles

- Les **boucles** :
- Exemples
 - `for (init-expr; cond-expr; incr-expr) bloc-d'instructions`
 - `for $x (liste) bloc-d'instructions`
 - `@list=('a','b','c','d','e','f') ;`
 - `for($e=0 ; $e <= $#list ; $e++) {`
 - `print $list[$e], "\n" ; # $e est un compteur, i.e. un entier`
 - `}`
 - `for $x (@list) {`
 - `print $x, "\n" # $x est élément de la liste, i.e. un caractère`
 - `}`

Les structures de contrôles

- Les **boucles** :
- Exemples
 - `for (init-expr; cond-expr; incr-expr) bloc-d'instructions`
 - `for $x (liste) bloc-d'instructions` (ou `foreach`)

 - `@list=('a','b','c','d','e','f');`
 - `for($e=0; $e <= $#list; $e++) {`
 `print $list[$e], "\n" ; # $e est un compteur, i.e. un entier`
 `}`
 `for $x (@list) {`
 `print $x, "\n" # $x est un élément de la liste , i.e. un caractère`
 `}`

 - `while (expr) { bloc d'instructions }`

Les subtilités pratiques de Perl

- Les variables par défaut (`$_`, `@_`, ...)
 - `for (@list) {`
 `print "$_ \n"; # $_ est un élément de la liste`
 `}`

Les subtilités pratiques de Perl

- Les variables par défaut (`$_`, `@_`, ...)
 - ```
for (@list) {
 print "$_ \n"; # $_ est un élément de la liste
}
```
- Les pragmas : des modules qui modifient le comportement de l'interpréteur. Deux à utiliser à permanence : `strict` (obligation de déclarer toutes les variables avec le mot-clé `my`) et `warnings` (afin d'être averti des problèmes potentiels dans le code)

```
!/usr/bin/perl
use strict;
use warnings;
```

```
#mon premier programme Perl
print "Hello World!\n";
```

# Les subtilités pratiques de Perl

- Les variables par défaut (`$_`, `@_`, ...)
  - `for (@list) {`  
    `print "$_\n";`   # `$_` est un élément de la liste  
    `}`
- Les pragmas : des modules qui modifient le comportement de l'interpréteur. Deux à utiliser en permanence : `strict` (obligation de déclarer toutes les variables avec le mot-clé `my`) et `warnings` (afin d'être averti des problèmes potentiels dans le code)

```
#!/usr/bin/perl
use strict ;
use warnings;
#mon premier programme Perl
print "Hello World!\n";
```

- Les expressions régulières : un motif qui décrit un ensemble de chaînes de caractères possibles ( `[^r]ond$` signifie toute chaîne de 4 lettres qui ne commence pas par `r`, se termine par «`ond`» et se trouve en fin de ligne)

# Exemples bioinfo

motif.pl donne les positions d'un motif recherché dans une séquence.

```
#!/usr/bin/perl
use strict;
use warnings;
my $motif = 'AGAA';
my $sequence =
'AACAAATTGAAACAATAAACAGAAACAAAAATGGA'.
'AGGCGATGAAAATCGAGAAGGATAACGCTCTCGAT';
my @positions;
for my $i (0..length $sequence) {
 if ($motif eq substr($sequence, $i, length $motif)) {
 push @positions, $i;
 }
}
print "Les positions trouvées sont : @positions.\n" ;
```

Exécution :

```
~$./motif.pl
Les positions sont : 20 50
```

# Les listes, piles, tableaux

Assigner une liste à un tableau :

```
my @planetes=('Mercure',Venus');
```

```
my @cop = @planetes; # copie par valeur
```

# Les listes, piles, tableaux

Assigner une liste à un tableau :

```
my @planetes=('Mercure',Venus');
my @cop = @planetes; # copie par valeur
```

Accès à un élément du tableau :

```
my $b = $planetes[0]; # 1ere case
my $a = $cop[$#cop]; # dernière case, i.e. taille-1 du table
```

# Les listes, piles, tableaux

Assigner une liste à un tableau :

```
my @planetes=('Mercure',Venus');
my @cop = @planetes; # copie par valeur
```

Accès à un élément du tableau :

```
my $b = $planetes[0]; # 1ere case
my $a = $cop[$#cop]; # derniere case, i.e. taille-1 du table
```

Parcours du tableau version 1 :

```
for ($i=0; $i<=$#planetes; $i++) {
 print $planetes[$i], " ";
}
```

# Les listes, piles, tableaux

Assigner une liste à un tableau :

```
my @planetes=('Mercure',Venus');
my @cop = @planetes; # copie par valeur
```

Acces a un element du tableau :

```
my $b = $planetes[0]; # 1ere case
my $a = $cop[$#cop]; # derniere case, i.e. taille-1 du tableau
```

Parcours du tableau version 1 :

```
for (my $i=0; $i<=$#planetes; $i++) {
 print $planetes[$i], " "; # $i numéro d'une case du tableau
}
```

Parcours du tableau version 2 :

```
foreach (@planetes) {
 Print $_, " "; # $_ élément dans une case du tableau
}
```



# Les listes, piles, tableaux

fonction push et pop (empiler/dépiler) :

```
push(@planetes,'Terre'); # ajout à droite : Mercure Venus Terre
```

# Les listes, piles, tableaux

fonction push et pop (empiler/dépiler) :

```
push(@planetes,'Terre'); # ajout à droite : Mercure Venus Terre
```

```
my $home = pop(@planetes); # enlève à droite :Mercure Venus et $home==Terre
```

```
push(@planetes,'Terre','Mars'); # ajout à droite : Mercure Venus Terre Mars
```

# Les listes, piles, tableaux

fonction push et pop (empiler/dépiler) :

```
push(@planetes, 'Terre'); # ajout à droite : Mercure Venus Terre
```

```
my $home = pop(@planetes); # enlève à droite :Mercure Venus et $home==Terre
```

```
push(@planetes, 'Terre', 'Mars'); # ajout à droite : Mercure Venus Terre Mars
```

syntaxe version 'mot-clé' (on enlève les parenthèse quand il n'y a pas d'ambiguïté) :

```
push @planetes 'Terre';
```

unshift et shift, idem push et pop mais cote gauche :

```
unshift @planetes 'Jupiter'; # ajout à gauche : Jupiter Mercure Venus Terre Mars
```

# Les tables de hachage

- Une table de hachage `%tab_hachage` est un tableau qui permet d'associer une valeur à une clé : `$tab_hachage{cle} = valeur`.

```
my %trois_vers_un = (
Ala=>A, Cys=>C, Asp=>D, Glu=>E,
Phe=>F, Gly=>G, His=>H, Ile=>I,
Lys=>K, Leu=>L, Met=>M, Asn=>N,
Pro=>P, Gln=>Q, Arg=>R, Ser=>S,
Thr=>T, Val=>V, Trp=>W, Tyr=>Y) ;
```

```
print "Le code pour l'Arginine(Arg) est $trois_vers_un{Arg}\n" ;
```

# Les tables de hachage

- Une table de hachage `%tab_hachage` est un tableau qui permet d'associer une valeur à une clé : `$tab_hachage{cle} = valeur`.

```
my %trois_vers_un = (
Ala=>A, Cys=>C, Asp=>D, Glu=>E,
Phe=>F, Gly=>G, His=>H, Ile=>I,
Lys=>K, Leu=>L, Met=>M, Asn=>N,
Pro=>P, Gln=>Q, Arg=>R, Ser=>S,
Thr=>T, Val=>V, Trp=>W, Tyr=>Y) ;
```

```
print "Le code pour l'Arginine(Arg) est $trois_vers_un{Arg}\n" ;
```

```
Assigner un element (crée le hachage s'il n'existe pas)
$trois_vers_un{"UKN"} = X;
```

# Les tables de hachage

- Une table de hachage `%tab_hachage` est un tableau qui permet d'associer une valeur à une clé : `$tab_hachage{cle} = valeur`.

```
my %trois_vers_un = (
Ala=>A, Cys=>C, Asp=>D, Glu=>E,
Phe=>F, Gly=>G, His=>H, Ile=>I,
Lys=>K, Leu=>L, Met=>M, Asn=>N,
Pro=>P, Gln=>Q, Arg=>R, Ser=>S,
Thr=>T, Val=>V, Trp=>W, Tyr=>Y) ;
```

```
print "Le code pour l'Arginine(Arg) est $trois_vers_un{Arg}\n" ;
```

```
Assigner un element (crée le hachage s'il n'existe pas)
```

```
$trois_vers_un{"UKN"} = X;
```

```
initialisation de plusieurs valeurs (version alternative)
```

```
my %hach = ("A",3,5.6,"C") # génère les paires "A"=>3 et 5.6=>"C"
```

```
my %cop = %hach; # copie par valeur
```

# Les tables de hachage

- Une table de hachage `%tab_hachage` est un tableau qui permet d'associer une valeur à une clé : `$tab_hachage{cle} = valeur`.

```
my %trois_vers_un = (
Ala=>A, Cys=>C, Asp=>D, Glu=>E,
Phe=>F, Gly=>G, His=>H, Ile=>I,
Lys=>K, Leu=>L, Met=>M, Asn=>N,
Pro=>P, Gln=>Q, Arg=>R, Ser=>S,
Thr=>T, Val=>V, Trp=>W, Tyr=>Y) ;
print "Le code pour l'Arginine(Arg) est $trois_vers_un{Arg}\n" ;
```

```
Assigner un element (crée le hachage s'il n'existe pas)
```

```
 $trois_vers_un{"UKN"} = X;
```

```
initialisation de plusieurs valeurs (version alternative)
```

```
 my %hach = ("A",3,5.6,"C") # génère les paires "A"=>3 et 5.6=>"C"
```

```
 my %cop = %hach; # copie par valeur
```

```
parcours par clé
```

```
 my @liste = keys(%hach); # my @liste = ("A", 5.6) ou (5.6, "A")
```

```
 for my $cle (keys(%hach)) {
```

```
 print "cle: $cle, valeur: $hach{$cle}\n";
```

```
 }
```

# Les tables de hachage

```
les valeurs seules
my @liste = values(%hach); # @liste = (3,"abc") ou ("abc",3)
for (values (%hach)) {
 print "valeur: $_\n";
}
```



# Les tables de hachage

# les valeurs seules

```
my @liste = values(%hach); # @liste = (3,"abc") ou ("abc",3)
foreach (values (%hach)) {
 print "valeur: $_\n";
}
```

# parcours des paires clé=>valeur

```
while (my ($cle,$val)=each(%hach)) {
 print "cle: $cle, valeur: $val\n";
}
```

# Les tables de hachage

```
les valeurs seules
my @liste = values(%hach); # @liste = (3,"abc") ou ("abc",3)
foreach (values (%hach)) {
 print "valeur: $_\n";
}

parcours des paires clé=>valeur
while (my ($cle,$val)=each(%hach)) {
 print "cle: $cle, valeur: $val\n";
}

pour détruire une paire
delete $hach{"A"}; # %hach ne contient plus "A"=>3
```

# Parcours d'un fichier

- Exemple de parcours ligne par ligne d'un fichier passé sur l'entrée standard du script Perl :

```
while (<>) {
 # traitement
 print $_ ;
}
```

- Exécution

```
~$./script.pl < file.txt
```

Par défaut, `$_` contient la ligne active

On peut nommer la variable explicitement : `while (my $ligne = <>) . . .`

# Manipulation de fichiers

- FILEHANDLE : un descripteur de fichier
- 3 descripteurs par défaut :
  - STDIN : entrée standard
  - STDOUT : sortie standard
  - STDERR : sortie d'erreur standard

# Manipulation de fichiers

- FILEHANDLE : un descripteur de fichier
- 3 descripteurs par défaut :
  - STDIN : entrée standard
  - STDOUT : sortie standard
  - STDERR : sortie d'erreur standard
- Pour les fichiers standards :
  - `open(FILE1, "nom_fichier") ; #ouverture en lecture seule`
  - `open(FILE2, ">nom_fichier") ; #ouverture en écriture (écrase le contenu existant)`
  - `open(FILE2, ">>nom_fichier") ; #ouverture en écriture (ajoute au contenu existant)`

# Manipulation de fichiers

- FILEHANDLE : un descripteur de fichier
- 3 descripteurs par défaut :
  - STDIN : entrée standard
  - STDOUT : sortie standard
  - STDERR : sortie d'erreur standard
- Pour les fichiers standards :
  - `open (FILE1, "nom_fichier") ; #ouverture en lecture seule`
  - `open(FILE2, ">nom_fichier") ; #ouverture en écriture (écrase le contenu existant)`
  - `open(FILE2, ">>nom_fichier") ; #ouverture en écriture (ajoute au contenu existant)`
- Fermeture d'un fichier
  - `close (FILE1) ;`

# Manipulation de fichiers

# Exemple :

```
open (EP, "/etc/passwd");
open (RES, ">res.txt");
while (<EP>) {
 chomp; #supprime le retour chariot à la fin de chaque ligne
 print RES "I saw $_ in passwd file.\n";
}
close (EP);
close(RES);
```

# gestion des exceptions

```
open (DATA, "rapport.txt") || die "Ne peut pas ouvrir rapport.txt : ".$!."\n";
```

# Expressions régulières

Pour trouver, extraire et remplacer une sous-chaîne, c'est le plus expressif des outils.

`m/$exprreg/` : opérateur de recherche (match). Le 'm' est facultatif.

`s/$exprreg/$chaine/` : opérateur de substitution (remplace mot à mot)

`tr/$caracteres/$Caracteres/` : opérateur de transcription (remplace lettre à lettre).

Utilise `$_` par défaut sinon il faut utiliser l'opérateur `=~`.



# Expressions régulières

Pour trouver, extraire et remplacer une sous-chaîne, c'est le plus expressif des outils.

`m/$exprreg/` : opérateur de recherche (match). Le 'm' est facultatif.

`s/$exprreg/$chaine/` : opérateur de substitution (remplace mot à mot)

`tr/$caracteres/$Caracteres/` : opérateur de transcription (remplace lettre à lettre).

Utilise `$_` par défaut sinon il faut utiliser l'opérateur `=~`.

Quelques exemples :

```
my $str="aaaaaaaaAaaaA";
print "ok!\n" if ($str =~ /a*Aa{3}A$/);
```

```
my $str="123456";
print "$1\n" if ($str =~ /123(\d+)6/);#on récupère le texte équivalent à l'expression entre parenthèse
```

```
my $str="abcdEqsgefFRGe";
$str =~ tr/a-z/ABCD/; # les lettres a, b, c et d deviennent respectivement A, B, C et D,
et toutes les lettres de e à z deviennent toutes D.
```

`$count = s/ab/AB/g` : renvoie le nombre total de substitutions possibles dans `$_` et modifie `$_`.

# Expressions régulières

```
my $sequence = "ATGCTAGCTAGCTa*CTGTAAGTCGATGC";
if ($sequence =~ /^[^N]*$/i) { # si le scalaire séquence ne contient pas 'N' ou 'n'
 print "La séquence est définie.\n";
}
my $motif = "CTA.C" ; # le '.' remplace n'importe quel caractère.
if ($sequence =~ /$motif/i) { # recherche non sensible à la casse de CTA.C
 print "motif '$motif' trouvé\n";
}
```

# Expressions régulières

```
my $sequence = "ATGCTAGCTAGCTa*CTGTAAGTCGATGC";
if ($sequence =~ /^[^N]*$/i) { # si le scalaire séquence ne contient pas 'N' ou 'n'
 print "La séquence est définie.\n";
}
my $motif = "CTA.C" ; # le '.' remplace n'importe quel caractère.
if ($sequence =~ /$motif/i) { # recherche non sensible à la casse de CTA.C
 print "motif '$motif' trouvé\n";
}
```

Que fait le code suivant :

```
my @h;
while(<>) {
 chomp;
 if (/^[a-z0-9]*:\w+:(\d+):(\d+):.*$/) {
 push @h, {'name' => $1, 'uid' => $2, 'gid'=>$3};
 }
}

foreach (@h) {
 print "$_->{'name'} : uid $_->{'uid'} gid $_->{'gid'}\n"
}
```

Si on l'utilise de cette façon : ./script.pl < /etc/passwd

# Petit jeu: trouver le nombre

Écrire un programme le plus court possible qui tire un nombre aléatoire entre 0 et 100 inclus et permet à l'utilisateur de le trouver (en indiquant plus petit ou plus grand). Le nombre d'essai est limité et il sera passé en paramètre au script.

# Petit jeu: trouver le nombre

Écrire un programme le plus court possible qui tire un nombre aléatoire entre 0 et 100 inclus et permet à l'utilisateur de le trouver (en indiquant plus petit ou plus grand). Le nombre d'essai est limité et il sera passé en paramètre au script.

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

```
use strict;
```

```
use v5.10;
```

```
my $steps = shift; #récupère l'argument de commande
```

```
my $num=int(rand(100));
```

```
while ($steps-- > 0 && $num != my $essai) {
```

```
 print "Saisir un nombre\n";
```

```
 my $essai = <STDIN>;
```

```
 $essai =~ /^^\d+$/ or die ("Pas un nombre");
```

```
 if ($num == $essai){
```

```
 print "Bravo! Ce nombre est bien ".$num."\n";
```

```
 break;
```

```
 }elsif ($num < $essai){
```

```
 print "plus petit\n";
```

```
 }else{
```

```
 print "plus grand\n";
```

```
 }
```

```
}
```

```
print "Nombre d'essais écoulé. Le nombre était $num.\n";
```

# Les fonctions

- Prototype  
sub nomsub  
  bloc d'instructions

# Les fonctions

- Prototype

sub nomsub

  bloc d'instructions

- Appel d'une fonction :

- `foo($a)` #appel d'une fonction avec un paramètre scalaire `$a` passé par valeur
- `foo $a` #appel d'une fonction avec un paramètre scalaire `$a` passé par valeur
- `foo(\@a)` #appel d'une fonction avec un paramètre qui est une référence vers la liste `@a`

# Les fonctions

- Prototype

sub noms

sub bloc d'instructions

- Appel d'une fonction :

- `foo($a)` #appel d'une fonction avec un paramètre scalaire `$a` passé par valeur
- `foo $a` #appel d'une fonction avec un paramètre scalaire `$a` passé par valeur
- `foo(@a)` #appel d'une fonction avec un paramètre qui est une référence vers la liste `@a`

- Retour des paramètres par valeurs
- Une fonction peut retourner plusieurs valeurs



# Les fonctions

- Prototype

sub noms

sub bloc d'instructions

- Appel d'une fonction :

- `foo($a)` #appel d'une fonction avec un paramètre scalaire `$a` passé par valeur
- `foo $a` #appel d'une fonction avec un paramètre scalaire `$a` passé par valeur
- `foo(@a)` #appel d'une fonction avec un paramètre qui est une référence vers la liste `@a`

- Retour des paramètres par valeurs

- Une fonction peut retourner plusieurs valeurs

- Les arguments sont implicitement dans la variable `@_`. On peut les récupérer avec :

```
my $p1 = shift ; #my permet de déclarer la variable et
```

```
my $p2 = shift ; #de la restreindre à la fonction. Sinon, visibilité globale
```

```
... ;
```

# Les références

- Quelques exemples sur les références

```
my @tab = (1, 2, 3);
my $ref1=\@tab; #référence vers un tableau existant
my $ref2=[1, 2, 3]; #référence vers un tableau anonyme
```

```
#parcours du tableau
foreach my $x (@$ref2) { #on déréférence
 print $_."\n";
}
```

```
#accès au valeurs via la référence
print "ok\n" if($ref1->[0] == $tab[0]); #on accède à la même donnée
print "ok2\n" if($ref1->[0] == $ref2->[0]); #même valeur
```

# Les fonctions

```
#!/usr/bin/perl
use strict;

my @l=(4,5,6);
print join(' ',carre(\@l))."\n"; #passage de la liste @l par référence

sub carre {
 my $l = shift; #on récupère la référence
 my @ret;
 foreach my $x (@$l) { #il faut indiquer à perl que la référence est en fait une liste
 push @ret, $x*$x;
 }
 return @ret; #retour par valeur de la liste @ret
}
```

# Les fonctions

Que fait le code suivant ?

```
#!/usr/bin/perl
use strict;

my @a=(1,2,3);
my @b=('a','b','c');
my @c=('d','e','f');
my @d=(4,5,6);

my @tailings = popmany(\@a, \@b, \@c, \@d);

print join(' ', @tailings)."\n";

sub popmany {
 my $aref;
 my @retlist=();
 foreach $aref (@_) {
 push @retlist, pop @$aref;
 }
 return @retlist;
}
```

# Les fonctions

Que fait le code suivant ?

```
#!/usr/bin/perl
use strict;

my @l=(4,5,6);
my ($carre, $sqr) = carre(\@l);
print "carré : ".(join(' ',@$carre))."\n";
print "sqrt : ".(join(' ',@$sqr))."\n";

sub carre {
 my $l = shift;
 my @ret;
 my @base;
 foreach my $x (@$l) {
 push @ret, $x*$x;
 push @base, sqrt($ret[$#ret])
 }
 \@ret,\@base;
}
```

# Accès à une base de données

```
#!/usr/bin/perl
use strict;
use DBI; #chargement du module DBI => interface générique pour l'accès à une base de données

connexion en utilisant le pilote DBI:Pg (database handle)
my $dbh = DBI->connect("DBI:Pg:dbname=login;host=dbserver", "login", "passwd", {'RaiseError' => 1});

CREATION d'une table
$dbh->do("create table test(a int primary key, b int)");

my $rows = $dbh->do("INSERT INTO test (a, b) VALUES (1, 2)");
$rows += $dbh->do("INSERT INTO test (a, b) VALUES (3, 4)");
print "$rows n-uplets insérés\n";

SELECTION (statement handle)
my $sth = $dbh->prepare("SELECT * FROM test");
$sth->execute();

on itère sur chaque ligne du résultat. fetchrow_hashref retourne une référence vers un hash avec le nom
de chaque attribut comme clé ou faux si rien à retourner
while(my $ref = $sth->fetchrow_hashref()) {
 print "$ref->{'a'}, $ref->{'b'}\n";
}

#on libère la mémoire (suppression du résultat de la requête)
$sth->finish;
print "\n";
```

# Accès à une base de données

# SELECT 2e version

```
my $sth = $dbh->prepare("SELECT * FROM test");
$sth->execute();
while (my @t=$sth->fetchrow_array()) { #fetchrow_array retourne une liste ou faux si rien à retourner
 print "$t[0], $t[1]\n";
}
$sth->finish;
print "\n";
```

# SELECT 3e version

```
my $sth = $dbh->prepare("SELECT * FROM test");
$sth->execute();
while (my $t=$sth->fetchrow_arrayref()) { #fetchrow_array retourne une référence vers
chaque n-uplet ou faux si rien à retourner
 print "$t->[0], $t->[1]\n";
}
$sth->finish;
```

# suppression de la table

```
$dbh->do("drop table test");
```

#fermeture de la connexion

```
$dbh->disconnect();
```

# Les modules et le CPAN

Le module CPAN (fourni avec PERL) permet l'installation d'autres modules.  
Tapez la ligne de commande dans un terminal:

```
perl -MCPAN -e 'shell'
```

Vous voyez alors s'afficher:

```
cpan[1]>
```

Pour installer un module, il faut taper :

```
install Module::Name
```

par exemple, pour installer un parseur de fichiers XML :

```
install XML::SIMPL ou install XML::Twig
```

Pour utiliser le module dans un script PERL :

```
use Module::Name ;
```