Perl

Thierry Lecroq

Université de Rouen
FRANCE

Plan

Introduction au langage Perl

Qu'est ce que PERL ?

P.E.R.L. signifie Practical Extraction and Report Language.
(« langage pratique d'extraction et d'édition »)
Créé en 1986 par Larry Wall (ingénieur système).
Au départ pour gérer un système de « News » entre deux réseaux.
C'est :

- un langage de programmation
- un logiciel gratuit (que l'on peut se procurer sur Internet notamment)
- un langage interprété :
  - pas de compilation
  - moins rapide qu'un programme compilé
  - chaque « script » nécessite d'avoir l'interpréteur Perl sur la machine pour s'exécuter

Introduction au langage Perl

Pourquoi Perl est devenu populaire

- portabilité : Perl existe sur la plupart des plateformes aujourd'hui (Unix, Linux, Windows, Mac, ...)
- gratuité : disponible sur Internet
- simplicité : quelques commandes permettent de faire ce que fait un programme de 500 lignes en C ou en Pascal
- robustesse : pas d'allocation mémoire à manipuler, chaînes, piles, noms de variables illimités...

Quelle utilisation ?

À l'origine Perl a été créé pour

1. manipuler des fichiers (notamment pour gérer plusieurs fichiers en même temps)
2. manipuler des textes (recherche, substitution)
3. manipuler des processus (notamment à travers le réseau)

⇒ Perl était essentiellement destiné au monde UNIX

Pourquoi utilise-t-on Perl aujourd'hui ?

1. conversion de formats de fichiers
2. générer, mettre à jour, analyser des fichiers HTML (notamment pour l'écriture de CGI)
3. accès « universel » aux bases de données

⇒ Perl n'est plus lié au monde UNIX

Quelle utilisation ?

Perl n'est pas fait pour

1. écrire des interfaces interactives (mais il existe le module tkperl)
2. le calcul scientifique (Perl n'est pas compilé : problème de performances)

Perl

Deux versions

- Perl 4 : L'ancienne version (encore très utilisée)
- Perl 5 : La nouvelle version qui intègre la programmation objet

Exemple

Petit programme poli :

```
#!/usr/bin/perl
print "Bonjour";
```

Perl

Exemple

Affichage du nombre de lignes d'un fichier :

```
#!/usr/bin/perl
open (F, '< monfichier') || die "Problème d'ouverture : $!" ;
my $i=0;
while (my $ligne = <F>) {
   $i++;
}
close F;
print "Nombre de lignes : $i\n";
```

`F` est un descripteur de fichier, que l'on peut appeler comme on veut (l'usage veut que l'on note les descripteurs de fichier en majuscules). Chaque instruction Perl se termine par un point-virgule.

Exécution

3 méthodes

- `/chemin/perl fichier.pl`
- `#/chemin/perl` en entête du fichier `fichier.pl` (ligne shebang)
- `/chemin/perl -e 'commande'`

Plan

Types de données

Constantes

- `1`
- `-12345`
- `1.6E16` (signifie 160 000 000 000 000 000)
- `'a'`
- `'bioinformatique'`

Types de données

Variables

Les variables sont précédées du caractère $

- `$i = 0 ;`
- `$c = 'a' ;`
- `$matiere = 'bioinformatique' ;`
- `$racine_carree_de_2 = 1.41421 ;`
- `$master = 'master de $matiere' ;`
  ⇒ 'master de bioinformatique'
- `$master = "master de $matiere" ;`
  ⇒ 'master de bioinformatique'

Attention : pas d'accents ni d'espaces dans les noms de variables
Par contre un nom peut être aussi long qu'on le veut

Tableaux, listes

Les tableaux peuvent être utilisés comme des ensembles ou des listes.
Toujours précédés du caractère `@`

```
@chiffres = (1,2,3,4,5,6,7,8,9,0);
@fruits = ('amande','fraise','cerise');
@alphabet = ('a'..'z');
@a = ('a'); @nul = ();
@cartes = ('01'..'10','Valet','Dame','Roi');
```

Les deux points `..` signifient de « tant à tant ».
On fait référence à un élément du tableau selon son indice par :
`$chiffres[1]` ($\Rightarrow$ `2`)
`$fruits[0]` ($\Rightarrow$ `'amande'`)

Remarque

En Perl (comme en C) les tableaux commencent à l'indice 0

Tableaux, listes

On peut affecter un tableau à un autre tableau :

```
@ch = @chiffres ;
@alphanum = (@alphabet, '_', @chiffres) ;
⇒ ('a','b,',...,'z','_',1,2,3,4,5,6,7,8,9,0)
@ensemble = (@chiffres, 'datte', 'kiwi', 12.45) ;
```

Tableaux, listes

Remarques

On dispose d'un scalaire spécial : `$#tableau` qui indique le dernier indice du tableau (et donc sa taille - 1) : `$fruits[$#fruits]` (⇒ `'cerise'`)
Possibilité de référencer une partie d'un tableau
`@cartes[6..$#cartes]` ⇒
`('07','08','09','10','Valet','Dame','Roi')`
`@fruits[0..1]` ⇒ `('amande', 'fraise')`

Tableaux associatifs

Ils sont toujours précédés du caractère % :

```
%prix = ('amande'=>30, 'fraise'=>9, 'cerise'=>25) ;
```

ou :

```
%prix = (amande=>30, fraise=>9, cerise=>25) ;
```

En Perl 4 la notation est :

```
%prix = ('amande',30, 'fraise',9, 'cerise',25) ;
```

On référence ensuite un élément du tableau par : `$prix{'cerise'}` (ou `$prix{cerise}`)

Exemple

```
%chiffre = ();
$chiffre{'un'} = 1;=> ou $chiffre{un} = 1;
print $chiffre{un};
$var = 'un'; print $chiffre{$var};
```

Perl 5 autorise les combinaisons, comme un tableau de tableaux

```
%saisons = (
  'abricot'=>['été'],
  'fraise'=> ['printemps','été'],
  'pomme'=>  ['automne','hiver'],
  'orange'=> ['hiver'],
  'cerise'=> ['printemps'],
  'amande'=> ['printemps','été','automne','hiver']);
```

ou

```
@departements = (
['Ain', 'Bourg-en-Bresse', 'Rhône-Alpes'],
['Aisne', 'Laon', 'Picardie'],
...
['Yonne', 'Auxerre', 'Bourgogne']);
```

Et l'on accédera à la région de l'Eure, ou à la préfecture de Seine-Maritime par `$departements[27 - 1][2]`, `$departements[76 - 1][1]`

Pas de booléens

On utilise des entiers sachant que 0 est évalué comme étant faux (en fait il s'agit de la chaîne de caractère vide) et 1 comme étant vrai.

Tableaux à deux dimensions

On peut utiliser les tableaux associatifs pour simuler des tableaux à 2 (ou n) dimensions :

```
%tableMultiplication = ('1,1'=>1, '1,2'=>2, ...,
'9,8'=>72,'9,9'=>81) ;

$traduction{'amande','anglais'} = 'almond';
$traduction{'amande','italien'} = 'amoria';
$traduction{'cerise','anglais'} = 'cherry';
```

On peut également utiliser les tableaux de tableaux de Perl 5 pour le faire :

```
@tableMultiplication = (
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],# Multiplié par 0
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9],# Multiplié par 1
[ 0, 2, 4, 6, 8,10,12,14,16,18],# Multiplié par 2
...
[ 0, 9,18,27,36,45,54,63,72,81]);# Multiplié par 9
```

On référencera alors 2*6 par `$tableMultiplication[6][2]`

Expressions

Opérateurs arithmétiques

`$a = 1; $b = $a; les variables a, et b auront pour valeur 1`
`$c = 53 + 5 - 2*4; => 50`

Plusieurs notations pour incrémenter une variable `$a = $a + 1 ;` ou `$a += 1 ;` ou encore `$a++ ;`
Même chose pour `*` (multiplication), `-` (soustraction), `/` (division), `**` (exponentielle) `$a *= 3 ; $a /= 2 ; $a -= $b ; ...`
modulo `(17 % 3=>2)`

Expressions

Chaînes de caractères

- concaténation

  `$c = 'ce' . 'rise' ;` ⇒ `$c` devient `'cerise'`

  `$c .= 's' ;` ⇒ `$c` devient `'cerises'`

- réplication

  `$b = 'a' x 5 ;` ⇒ `'aaaaa'`

  `$b = 'jacqu' . 'adi' x 3 ;` ⇒ `'jacquadiadiadi'`

  `$b = 'assez ! ' ; $b x= 5 ;` ⇒ `'assez ! assez ! assez ! assez ! assez !'`

Expressions

Parenthèses

Comme dans tous les langages de programmation, les parenthèses peuvent être utilisées dans les expressions : `$x = 2 * (56 - 78) ;`

Expressions

Comparaisons

- de chiffres

  opérateurs habituels : `>, >=, <, <=, ==, !=`

  Attention : `=` est une affectation, `==` est une comparaison

- de chaînes

  `gt, ge, lt, le, eq, ne`

  Attention ! Ne pas confondre la comparaison de chaînes et d'entiers

  `'b' == 'a'` $\Rightarrow$ évalué comme étant vrai !

  il faut écrire :

  `'b' eq 'a'` $\Rightarrow$ évalué faux bien-sûr

Expressions

Comparaisons

- de booléens
  Même si le type booléen n'existe pas en tant que tel, des opérateurs existent : || (ou inclusif), && (et), ! (négation)
  `( ! 2 < 1)` ⇒ vrai `(1 < 2) && (2 < 3)` ⇒ vrai `($a < 2) || ($a == 2)` équivaut à `($a <= 2)`
  `( !$a && !$b)` équivaut à `!($a || $b)` (règle de Morgan !)
  Remarque : depuis Perl 5 une notation plus agréable existe : `or` (au lieu de ||), `and` (au lieu de &&), `not` (au lieu de !)
  ```
  if (not ($tropCher or $tropMur)) print "J'achete !" ;
  ```

Syntaxe générale

Les commentaires commencent par un #.
Tout le reste de la ligne est considéré comme un commentaire.
Un bloc est un ensemble d'instructions entourées par des crochets ({}), chaque instruction étant suivie d'un point-virgule.

Plan

Expressions conditionnelles

si

```
if ($prix{'datte'} > 20) {
   print 'Les dattes sont un peu chères...';
   print 'Achetons plutôt des cerises';
}

if ($fruit eq 'fraise') {
   print 'parfait !';
} else {
   print 'Bof !';
   print 'Je préfère sans pépin';
}
```

Expressions conditionnelles

si

```
if (($fruit eq 'cerise') or ($fruit eq 'fraise')) {
   print 'rouge';
} elsif ($fruit eq 'banane') {
   print 'jaune';
} elsif ($fruit eq 'kiwi') {
   print 'vert';
} else {
   print 'je ne sais pas...';
}
```

Expressions conditionnelles

Remarque : il existe une autre notation : `commande if (condition) ;`

```
print 'Quand nous chanterons...' if ($fruit eq 'cerise') ;
```

Condition inversée :

```
unless (condition) {
   bloc
};
```

Ou, selon la notation alternative : `commande unless (condition) ;`

```
print 'Avec du sucre SVP...' unless ($fruit eq 'fraise') ;
```

Boucles

tant que

```
my $monArgent = $porteMonnaie;
while ($monArgent > $prix{'cerise'}) {
    $monArgent -= $prix{'cerise'};
    print "Et un kilo de cerises !";
}
```

Remarque : là aussi il existe une autre notation : `commande while (condition)`

```
print "Je compte : $i" while ($i++ < 10);
```

Boucles

répéter

```
# Recherche du premier fruit <= 10 euros
my $i = 0; my $f;
do {
    $f = $fruits[$i];
    $i++;
} while ($prix{$f} > 10);
print "Je prend : $f";

my $i = 10;
do {
    print $i;
    $i--;
} until ($i == 0);
```

Boucles

pour

```
for (my $monArgent = $porteMonnaie;
        $monArgent > $prix{'cerise'};
        $monArgent -= $prix{'cerise'}) {
   print "Et un kilo de cerises !";
}

for (my $i=0; $i <= $#fruit; $i++) {
   print $fruit[$i];
}
```

Attention à bien mettre un point-virgule entre les différents éléments d'un `for`

Boucles

pour tout

```
foreach my $f (@fruits) {
    print $f;
}
```

Plan

Procédures et fonctions

Déclaration

```
sub maProcedure {
   bloc;
}
```

Appel : `&maProcedure ;` ou (depuis Perl 5) : `maProcedure ;`

avec paramètre(s)

```
sub pepin {
   my($fruit) = @_;
   if (($fruit ne 'amande') and ($fruit ne 'fraise')) {
      print "ce fruit a des pépins !";
   }
}
```

Appel : `&pepin('cerise') ;` ou (depuis Perl 5) : `pepin('cerise') ;`

Fonctions

```
sub pluriel {
    my($mot) = @_;
    $m = $mot.'s';
    return($m);
}
```

Appel : `$motAuPluriel = &pluriel('cerise');`

$\Rightarrow$ `'cerises'`

Procédures et fonctions

Remarque

Le passage de paramètres se fait donc à l'aide du tableau spécial `@_` (géré par le système Perl).
L'instruction `my` réalise une affectation dans des variables locales à la procédure avec les éléments du tableau.
Ce type de passage de paramètre est très efficace car le nombre de paramètres n'est pas forcément fixe.

Procédures et fonctions

Variables locales

Attention au petit piège habituel :

```
$m = 'Dupont';
$f = &pluriel('cerise');
print "M. $m vend des $f\n";
```

affichera : `M. cerises vend des cerises !`
À cause du fait qu'on utilise la variable `$m` qui est modifiée dans la fonction `pluriel` ...
La solution consiste à déclarer toutes les variables utilisées dans les procédures en variables locales à l'aide de `my`.
D'ou l'avantage d'utiliser les modules `strict` et `warnings`.
À la troisième ligne de la fonction `pluriel` on rajoutera : `my $m ;`

Fonctions prédéfinies

Quelques fonctions offertes par Perl pour manipuler les données.
L'inventaire n'est pas exhaustif.

Système

- `print` : permet d'afficher un message, ou le contenu de variables.

  ```
  print 'bonjour';
  print 'J\'ai acheté ', $nombre,' kilos de ', $fruit;
  ```

  `print ;` ⇒ affiche le contenu de la variable spéciale `$_`
  ou encore :

  ```
  print "J'ai acheté $nombre kilos de ",
  &pluriel($fruit) ;
  ```

  Quelques caractères spéciaux affichables avec `print` : `\n` ⇒ retour-chariot, `\t` ⇒ tabulation, `\b` ⇒ bip

- `exit` : permet d'arrêter le programme en cours `if ($erreur)`

  ```
  exit ;
  ```

Fonctions prédéfinies

Système

- `die` : permet d'arrêter le programme en cours en affichant un message d'erreur.
  ```
  if ($fruit eq 'orange') die 'Je déteste les oranges !'
  ```
- `system` : permet de lancer une commande système
  ```
  system 'mkdir monRepertoire' ;
  ```
- `sleep n` : le programme dort pendant n secondes
  ex : programme « bip horaire »
  ```
  while (1) sleep 3600 ; print "\a" ;
  ```
  le fait d'écrire `while (1)` permet de faire une boucle infinie (on aurait pu écrire `for ( ; ;)`)

Fonctions prédéfinies

Mathématique

Les fonctions mathématiques habituelles existent aussi en Perl : `sin`, `cos`, `tan`, `int` (partie entière d'un nombre), `sqrt`, `rand` (nombre aléatoire entre 0 et n), `exp`, `log`, `abs`.

```
$s = cos(0) ;
```
$\Rightarrow 1$

```
$s = log(exp(1)) ;
```
$\Rightarrow 1$

```
$i = int(sqrt(8)) ;
```
$\Rightarrow 2$

```
$tirageLoto = int(rand(49)) + 1 ;
```

```
$i = abs(-5.6)
```
$\Rightarrow 5.6$

Fonctions prédéfinies

Chaînes de caractères

- `chop(ch)` : enlève le dernier caractère de la chaîne
  `$ch='cerises' ; chop($ch) ;` ⇒ `ch` contient `'cerise'`
- `chomp(ch)` : même chose que `chop` mais enlève uniquement un retour-chariot éventuel en fin de chaîne.
- `length(ch)` : retourne la longueur de la chaîne (nombre de caractères)
  `$l = length('cerise')` ⇒ 6
- `uc(ch)` : retourne la chaîne en majuscules (Perl 5)
  `$ch = uc('poire')` ⇒ `'POIRE'`
- `lc(ch)` : retourne la chaîne en minuscules (Perl 5)
  `$ch = lc('POIRE')` ⇒ `'poire'`
- `lcfirst(ch)`, `ucfirst(ch)` retourne la chaîne avec simplement le premier caractère en minuscule/majuscule (Perl 5)
  `$ch = ucfirst('la poire')` ⇒ `'La poire'`

Fonctions prédéfinies

Chaînes de caractères

- `split('motif', ch)` : sépare la chaîne en plusieurs éléments (le séparateur étant motif)
  Le résultat est un tableau. (Fonction très utilisée)
  `@t = split(' / ', 'amande / fraise / cerise');`
  ⇒ `('amande','fraise', 'cerise')`
- `substr(ch, indiceDébut, lg)` : retourne la chaîne de caractères contenue dans `ch`, du caractère `indiceDébut` et de longueur `lg`
  `$ch=substr('dupond', 0, 3)` ⇒ `'dup'`
  `$ch=substr('Les fruits', 4)` ⇒ `'fruits'`
- `index(y, x)` : retourne la position de `x` dans la chaîne `y`
  `$i=index('Le temps des cerises','cerise');` ⇒ `13`

Remarque

Par défaut la plupart de ces fonctions travaillent sur la variable spéciale `$_`
`= 'amandes'; chop; print;` ⇒ Affichera `'amande'`

Fonctions prédéfinies

Tableaux, listes

- `grep(/expression/, tableau)` : recherche d'une expression dans un tableau
  `if (grep(/poivron/, @fruits)) ;` ⇒ `faux`
  `if (grep(/$f/, @fruits) print 'fruit connu' ;`
  `grep` retourne un tableau des éléments trouvés :
  `@f = grep(/ise$/, @fruits) ;` ⇒ `fraise ;cerise`
- `join(sep, tableau)` : regroupe tous les éléments d'un tableau dans une chaîne de caractères (en spécifiant le séparateur)
  `print join(', ', @fruits) ;` ⇒ affiche `'amande, fraise, cerise'`

Fonctions prédéfinies

Tableaux, listes

- `pop (tableau)` : retourne le dernier élément du tableau (et l'enlève)
  `print pop(@fruits) ;` ⇒ affiche `'cerise'`, `@fruits` devient `('amande','fraise')`
- `push (tableau, element)` : ajoute un élément en fin de tableau (contraire de `pop`)
  `push(@fruits, 'abricot') ;` ⇒ `@fruits` devient `('amande','fraise','abricot')`
- `shift(tableau)` : retourne le premier élément du tableau (et l'enlève)
  `print shift(@fruits)` ⇒ affiche `'amande'`, `@fruits` devient `('fraise','abricot')`
- `unshift (tableau, element)` : ajoute un élément en début de tableau
  `unshift ('coing', @fruits) ;` ⇒ `@fruits` devient `('coing', 'fraise','abricot')`

Fonctions prédéfinies

Tableaux, listes

- `sort (tableau)` : tri le tableau par ordre croissant
  `@fruits = sort (@fruits) ;` ⇒ `@fruits` devient `('abricot', 'coing', 'fraise')`
- `reverse (tableau)` : inverse le tableau
  `@fruits = reverse(@fruits) ;` ⇒ `@fruits` devient `('fraise', 'coing', 'abricot')`
- `splice (tableau, début, nb)` : enlève `nb` éléments du tableau à partir de l'indice `début`
  `@derniers = splice(@fruits, 1,2) ;`
  ⇒ `@derniers` devient `('coing', 'abricot')`, `@fruits` devient `('fraise')`
  On peut éventuellement remplacer les éléments supprimés :
  `@fruits=('fraise','pomme') ;`
  `splice(@fruits, 1,1, ('elstar', 'golden')) ;`
  ⇒ `@fruits` contient `('fraise', 'elstar', 'golden')`

Exemple : Tirage des 5 chiffres du loto

```
my @c = (1..49); my $i=0;
print splice(@c, int(rand($#c+1)),1),"\n" while ($i++ < 5);
```

On enlève 5 fois un élément (pris au hasard) du tableau des 49 chiffres.
Pour information, l'exemple précédent aurait pu s'écrire :

```
my @c = (1..49); my $i=0;
while ($i < 5) {
   my $nb = int(rand($#c + 1));
   print "$nb\n";
   splice (@c, $nb, 1);
   $i++;
}
```

Fonctions prédéfinies

Tableaux associatifs

- `each(tabi)` : les couples clé/valeurs d'un tableau associatif

```
while (my ($fruit, $valeur) = each(%prix)) {
   print "kilo de $fruit : $valeur F";
}
```

L'utilisation habituelle est d'afficher le contenu d'un tableau.
Attention : Les clés d'un tableau associatif ne sont pas triées !
ex :

```
my %t = ("bernard"=>45, "albert"=>32, "raymond"=>2);
while (my ($n, $s) = each(%t)) {print "$n,$s\n";}
```

affichera :

```
raymond,2
albert,32
bernard,45
```

Fonctions prédéfinies

Tableaux associatifs

- `values(tabi)` : toutes les valeurs d'un tableau associatif (sous la forme d'un tableau)

```
print 'les prix :', join(', ', values(%prix)) ;
```

- `keys(tabi)` : toutes les clés d'un tableau associatif

```
print 'les fruits :', join(', ', keys(%prix)) ;
```

- `exists(élément)` : indique si un élément a été défini

```
if (exists $prix{'kiwi'}) {
   print $prix{'kiwi'};
} else {
   print 'Je ne connais pas le prix du kiwi !';
}
```

- `delete(élément)` : supprimer un élément

```
delete $prix{'cerise'} ;
```

Fonctions prédéfinies

Remarque

Il n'existe pas de fonction permettant d'ajouter un élément dans un tableau associatif (comme le `push` des tableaux normaux) car il suffit d'écrire :

```
$tab{nouvel-élément} = nouvelle-valeur ;
```

Plan

Gestion de fichiers

Ouverture en lecture

L'ouverture consiste (le plus souvent) à associer un descripteur de fichier (`filehandle`) à un fichier physique.
`open(FENT, '< fichier') ;` : ouverture d'un fichier, référencé ensuite par `FENT`
Le caractère `<` est facultatif mais recommandé
`open(FENT, 'fruits.txt') ;`
Depuis Perl 5.8, on peut ouvrir un fichier en spécifiant l'encodage utilisé :

```
open(FENT, '<:encoding(iso-8859-1)', 'fruits.txt');
open(FENT, '<:encoding(iso-8859-7)', 'texteGrec.txt');
open(FENT, '<:utf8', 'texteUnicode.txt');
```

`open(COM, 'commande|') ;` : ouverture d'une commande dont le résultat sera dans `COM`
`open(FMASTER, 'ls /users/master |') ;` (uniquement sous Linux)
Un fichier spécial : `STDIN`, le clavier (entrée standard).

Gestion de fichiers

Ouverture en écriture

`open(FSOR, '> fichier') ;` écriture du fichier, si ce fichier existait auparavant : l'ancien contenu est écrasé.

`open(FSOR, '> etat.txt') ;`

`open(FSOR,'>>fichier') ;` : écriture à la fin du fichier, Le fichier est créé si besoin

`open(FSOR, '>> liste.txt') ;`

`open(FSOR, '| commande') ;` : le fichier de sortie sera en fait l'entrée standard de la commande

`open(FIMP, '| lpr') ;` : la sortie sera imprimée

`open(FMAIL, '| mail -s "Bonjour" Thierry.Lecroq@univ-rouen.fr') ;`

Deux fichiers spéciaux : `STDOUT`, `STDERR` (respectivement sortie standard et sortie erreur), par défaut l'écran.

Gestion de fichiers

Gestion des erreurs

Lorsque l'on ouvre un fichier il se peut qu'il y ait une erreur.
En lecture : le fichier n'existe pas, ou ne peut pas être lu (droits d'accès)...
En écriture : Le fichier existe mais on n'a pas le droit d'y écrire ; pour une commande Linux : la commande est inconnue...
Il faut prendre l'habitude, quand on ouvre un fichier, de détecter l'erreur éventuelle.
On peut le faire sous la forme suivante : (dérivée du C)

```
if (! open (F, ...)) {
   die "Problème à l'ouverture du fichier";
}
```

Ou sous la forme plus simple et plus usitée en Perl :

```
open (F, ...) || die "Pb d'ouverture" ;
```

On peut, et c'est même conseillé, récupérer le texte de l'erreur contenu dans la variable `$!`

```
open (F, ...) || die "Pb d'ouverture : $!" ;
```

Gestion de fichiers

Fermeture

Commande close

```
close FENT ; close FSOR ;
```

Lecture

```
$ligne = <FENT> ;
```

```
$reponse = <STDIN> ;
```
: lecture d'une ligne au clavier

Remarques :

- La fin de ligne (retour-chariot) est lue également. Pour enlever cette fin de ligne il suffit d'utiliser la commande `chop`, ou son équivalent `chomp` (enlève le dernier caractère uniquement si c'est un retour-chariot)
- On peut lire toutes les lignes d'un fichier dans un tableau (en une seule instruction)

```
@lignes = <FENT> ;
```

Gestion de fichiers

Écriture

```
print FSOR ce-que-je-veux;
print FSOR 'DUPONT Jean';
print FMAIL 'Comment fait-on pour se connecter SVP ?';
print STDOUT "Bonjour\n";
print STDERR 'Je déteste les oranges !';
```

Parcours

Se fait d'une manière intuitive :

```
open (F, $fichier) || die "Problème pour ouvrir $fichier: $!";
while (my $ligne = <F>) {
   print $ligne;
}
close F;
```

L'instruction `$ligne = <F>` retourne toujours quelque chose sauf à la fin du fichier, la condition devient alors fausse.

Gestion de fichiers

Fichier spécial

`<>`

Perl offre une fonctionnalité bien pratique : l'utilisation d'un fichier spécial en lecture qui contiendra ce qui est lu en entrée standard.
Exemple : le petit programme du début peut s'écrire

```
#!/usr/bin/perl
my $i=0;
while (my $ligne = <>) {
   $i++;
}
print "Nombre de lignes : $i\n";
```

Remarque : Lorsque, dans une boucle `while`, on ne spécifie pas dans quelle variable on lit le fichier : la ligne lue se trouvera dans la variale spéciale `$_`.
Ainsi ce programme demandera d'écrire quelques lignes et affichera le nombre de lignes qu'il a lu. Mais il permet également de prendre un ou plusieurs fichiers en paramètre.

Plan

Expressions régulières

Une expression régulière (appelée aussi « regexp ») est un motif de recherche, constitué de :

- Un caractère
- Un ensemble de caractères :
  - `[a-z]` : tout caractère alphabétique
  - `[aeiouy]` : toute voyelle
  - `[a-zA-Z0-9]` : tout caractère alphanumérique
  - le caractère `^` au début d'un ensemble signifie « tout sauf »(`[^0-9]` : tout caractère non numérique)
  - Un caractère spécial : `\n` (retour-chariot), `\t` (tabulation), `^` (début de ligne), `$` (fin de ligne)
  - `\w` signifie « un mot », `\s` signifie « un espace », `\W` « tout sauf un mot », `\S` « tout sauf un espace »
- Quelques opérateurs : `?` (0 ou 1 fois), `*` (0 ou n fois), `+` (1 ou n fois), `|` (ou inclusif), `n,m` (de n à m fois)

Expressions régulières

Remarque

Si l'on veut qu'un caractère spécial apparaisse tel quel, il faut le précéder d'un antislash \.
Les caractères spéciaux sont :

```
^ | ( ) [ ] { } / \ } * ? +
```

`if (/\(.*\)/) print ;` : expression contenant des parenthèses

Attention

Le caractère spécial ^ a deux significations différentes :

1. dans un ensemble il signifie « tout sauf »
2. en dehors il signifie « commence par »

Expressions régulières

Comme la commande `egrep` d'Unix/Linux, Perl offre la même puissance de manipulation d'expressions régulières.
On utilise l'opérateur conditionnel `=~` qui signifie « ressemble à ».
Syntaxe : `chaîne =~/expression/`
Exemple : `if ($nom =~ /^[Dd]upon/) print "OK !";`
⇒ Ok si nom est `'dupont'`, `'dupond'`, `'Dupont-Lassoeur'`
`^` signifie « commence par »
On peut rajouter `i` derrière l'expression pour signifier qu'on ne différencie pas les majuscules des minuscules.
Le contraire de l'opérateur `=~` est `!~` (ne ressemble pas à ...)

```
if ($nom !~ /^dupon/i) {print "Non...";}
```

Expressions régulières

Exemples d'utilisation

Le traitement d'une réponse de type (Oui/Non) : en fait on teste que la réponse commence par « O »

```
print 'Etes-vous d\'accord ? ';
my $reponse = <STDIN>;
if ($reponse =~ /^O/i) { # commence par O (minus. ou majus.)
   print "Alors on continue";
}
```

Remarque : le petit piège dans ce programme serait de mettre `$reponse =~ /O/` (sans le ^) qui serait reconnu dans `NON` (qui contient un `O`)

Expressions régulières

Exemples d'utilisation

Recherche des numéros de téléphone dans un fichier :

```
while (my $ligne = <>) { # Tant qu'on lit une ligne
   if ($ligne =~ /([0-9][0-9]\.)+[0-9][0-9]/) {
      # si la ligne est sous la forme 02.99.45...
      print $ligne; # Alors on l'affiche
   }
}
```

Remarque : par défaut l'expression régulière est appliquée à la variable `$_`.
Donc écrire `if ($_ =~ /exp/)` est équivalent à écrire : `if (/exp/)`

Expressions régulières

Exemples d'utilisation

Utilisation avancée :
Le contenu des parenthèses d'une expression régulière est retourné sous la forme d'un tableau.
Exemple : Lister tous les hyperliens d'un document HTML
Un hyperlien est noté sous la forme `<a href="quelquechose">` (avec quelquechose ne pouvant pas contenir le caractère ".
Donc quelquechose sera reconnu par

```
[^\"]
```

qui signifie tout caractère autre que ".

```
while (my $ligne = <>) {
   if ( my ($h) = ($ligne =~ /<a href=\"([^\"]+)\">/)) {
      print "Hyperlien : $h\n";
   } # ce programme ne détecte qu'un hyperlien par ligne
}
```

Expressions régulières

On peut utiliser cette fonctionnalité pour faire des affectations :
Exemple : on a un fichier annuaire de la forme :

```
M. Dupont tél : 02.99.27.82.67
M. Durant tél : 02.99.34.54.56
...
Mme Larivoisière 02.99.23.43.21
```

Et on souhaiterait avoir un programme automatique qui soit capable de retrouver un numéro de téléphone d'après un nom (saisi au clavier).

Expressions régulières

```
print " entrez un nom : ";
my $nom = <STDIN>; # on lit ce que « tape » l'utilisateur
chomp($nom); # Enlève le « retour-chariot » en fin de chaîne
open (ANNUAIRE, '<mon-annuaire.txt') || die "Problème ouvertu
while (my $li = <ANNUAIRE>) {
   if (my ($tel) = ($li =~ /$nom.*(([0-9][0-9]\.)+[0-9][0-9])/
      print "Le numéro de téléphone de $nom est : $tel\n";
   }
}
close ANNUAIRE;
```

Expressions régulières

Remplacement

Comme le fait la commande `sed` en Unix, Perl permet de faire des remplacements sur une chaîne de caractère, en utilisant la syntaxe :

```
$chaine =~ s/motif/remplacement/ ;
```

où `motif` est une expression régulière et `remplacement` ce qui remplace.

Exemple : `$fruit =~ s/e$/es/ ;` : remplace un `e` final par `es`

On peux référencer une partie du motif dans le remplacement avec `$1` (`$1` est une variable spéciale : le contenu de la première parenthèse).

Exemple :

Transformer automatiquement les noms d'arbre par « arbre à fruit »

```
$texte =~ s/([a-z]+)ier /arbre à $1es / ;
```

`'cerisier'` sera transformé en `'arbre à cerises'`

`'manguier'` : `'arbre à mangues'`

(contenu de `$1` : `'mangu'`)

Expressions régulières

Remplacement

Les options :

`s/exp/remp/i ;` : indifférenciation minuscules/majuscules

`s/exp/remp/g ;` : remplace toute occurrence (pas seulement la première)

Pour remplacer un texte par le même en majuscule (`\U`) :

```
\s/([a-z])/\U$1/g;
```

Exemple : mettre toutes les balises d'un fichier HTML en majuscule (en fait on met le premier mot de la balise en majuscule).

Ne pas oublier les balises de fin (commençant par `/`)

```
while (my $ligne = <>) {
  $ligne =~ s/\<(\/?[a-z]+)/\<\U$1/g;
  # \U$1 signifie $1 en majuscules
  print $ligne;
}
```

Expressions régulières

Remplacement

Si on appelle ce programme avec le fichier HTML suivant :

```
<html><body bgcolor="ffffff">
<a href="http://chl.univ-mlv.fr/">bonjour</a>
</body></html>
```

Il affichera :

```
<HTML><BODY bgcolor="ffffff">
<A href="http://chl.univ-mlv.fr/">bonjour</A>
</BODY></HTML>
```

Expressions régulières

L'opérateur `tr` permet d'effectuer des remplacements simultanés de caractères.
`$variable =~ tr/`$c_0c_1 \ldots c_m$`/`$d_0d_1 \ldots d_n$`/ ;`

- Si $m \leq n$ alors transforme simultanément chaque occurrence de c_i dans `$variable` par d_i (pour $0 \leq i \leq m$).
- Si $m > n$ alors transforme simultanément chaque occurrence de c_i dans `$variable` par d_i (pour $0 \leq i \leq n$) et transforme simultanément chaque occurrence de c_j dans `$variable` par d_n (pour $n+1 \leq j \leq m$).

Cette instruction retourne le nombre de substitutions effectuées.
Si $d_0d_1 \ldots d_n$ est vide alors compte le nombre total d'occurrences des $c_0c_1 \ldots c_m$ dans `$variable`.

Plan

Variables spéciales

Variables de la forme `$c` (avec `c` un caractère non alphabétique) :

- `$_` : dernière ligne lue (au sein d'une boucle `while`)
- `$!` : dernière erreur, utilisée dans les détections d'erreurs `open(F,'fichier') || die "erreur $!"`
- `$!` : numéro système du programme en cours, parfois utile car il change à chaque fois
- `$1`, `$2`, ... : contenu de la parenthèse numéro n dans la dernière expression régulière
- `$0` : nom du programme (à utiliser, cela vous évitera de modifier le programme s'il change de nom)

Tableaux spéciaux

- `@_` : paramètres passés à une procédure
- `@ARGV` : paramètres passés au programme
- `%ENV` : tableau associatif contenant toutes les variables d'environnement
- `@INC` : tous les répertoires Perl contenant les librairies (rarement utilisé)

Structures complexes

En Perl (version 5), on utilise la syntaxe des tableaux associatifs pour désigner une structure complexe, exemple :

```
$patient->{nom} = 'Dupont';
$patient->{prenom} = 'Albert';
$patient->{age} = 24;
```

On peut initialiser la structure comme lorsqu'on initialise un tableau associatif :

```
$patient = {
nom => 'Dupont',
prenom => 'Albert',
age => 24};
```

On utilise les accolades plutôt que les parenthèses (référence vers un tableau associatif)

Structures complexes

Pour afficher le contenu d'une structure complexe :

```
print "Nom : $patient->{nom} $patient->{prenom},
âge : $patient->{age}";
```

ou :

```
foreach my $champ ('nom','prenom','age') {
   print "$champ : $patient->{$champ}\n";
}
```

qui affiche :

```
nom : Dupont
prenom : Albert
age : 24
```

Structures complexes

ou encore :

```
foreach my $champ (keys %$patient) {
   print "$champ : $patient->{$champ} ";
}
```

qui affiche :

```
prenom : Albert
nom : Dupont
age : 24
```

Structures complexes

Passages de paramètres au programme

On peut appeler un programme Perl en lui donnant des paramètres, comme on le fait pour les commandes Unix
Les paramètres sont stockés dans un tableau spécial : `@ARGV`
Le premier paramètre est donc `$ARGV[0]`
exemple : `bienvenue.pl`

```
#!/usr/bin/perl
use strict; use warnings;
my $email = $ARGV[0];
open (MAIL, "| mail -s 'Bonjour' $email") # commande unix mail
|| die "Problème : $!";
print MAIL "Bonjour très cher\n";
print MAIL "Nous sommes très heureux de vous envoyer un mail\n"
print MAIL "A bientôt\n"
close MAIL;
```

L'appel (sous Unix/Linux) se fera : `bienvenue.pl`
Thierry.Lecroq@univ-rouen.fr

Structures complexes

Parcours de fichiers

```
#!/usr/bin/perl
while (my $ligne = <>) {
   print $ligne;
}
```

Qui se contente de lire le contenu du (ou des) fichier(s) passé(s) en paramètre pour l'afficher à l'écran.
On peut avoir envie d'écrire le résultat d'une modification dans un fichier, par exemple, enlever toutes les balises HTML et les écrire dans un fichier `"sansbalises.txt"` (ouvert donc en écriture).

Structures complexes

Parcours de fichiers

```
#!/usr/bin/perl
use strict; use warnings;
open (ST, '> sansbalises.txt') ||die "Impossible d'écrire: $!"
while (my $ligne = <>) {
   ligne =~ s/<[^>]+>//g;# On remplace <...> par rien du tout
   print ST $ligne; # Le résultat est écrit dans le fichier
}
close ST;
```

Par contre le traitement serait relativement complexe pour prendre tous les fichiers un par un, et, à chaque fois, lui enlever les balises, Perl a une option (`-i`) qui permet de renommer le fichier d'origine et le résultat de la transformation sera écrite dans le fichier lui-même.
Cela parait compliqué mais c'est bien utile pour faire des transformations automatiques sur plusieurs fichiers en même temps.

Structures complexes

Parcours de fichiers

Cette option (`-i extension`) se met dans la première ligne :

```
#!/usr/bin/perl -i.avecbalises
while (my $ligne = <>) {
   $ligne =~ s/<[^>]+>//g;
   print $ligne; # Affichage dans un fichier, pas à l'écran
}
```

Ainsi on peut lui donner en paramètre tous les fichiers html d'un répertoire, ils seront tous renommés sous la forme fichier.html.avecbalises, et le fichier html de départ sera le résultat du programme.

Structures complexes

Parcours de fichiers

Remarque :
Sous Unix/Linux, pour tester une petite commande Perl, on peut appeler Perl sans créer de programme avec la syntaxe : `perl -e 'commande'`
Ou même on peut l'utiliser pour appliquer une commande à chaque ligne :

```
ls | perl -n -e 'print if (! /truc/);'
```

$\Rightarrow$ affiche tous les éléments du répertoire courant sauf truc
Quand on utilise les option `n` et `e`, chaque ligne lue se trouve dans la variable spéciale `$_`
l'option `-e` permet de prendre une commande Perl en paramètre
l'option `-n` permet d'appliquer la commande à chaque ligne lue
l'option `-iext` permet de renommer un fichier avec l'extension `ext` avant d'appliquer des traitements
Exemple :
Supprimer en une seule commande toutes les balises html de plusieurs fichiers :

```
perl -n -i.avecbalises -e 's/<[^>]+>//g ; print ;'
```

Car l'option `-i` permet de renommer chaque fichier (avec l'extension donnée), ensuite, tout ce qui est lu est lu dans chaque fichier, tout ce qui est écrit est écrit dans chacun des fichiers.
Autre exemple :
Enlever toutes les lignes des fichiers qui comporte la chaîne de caractères `secret`

```
perl -n -i.avecbalises -e 'print unless (/secret/) ;'
```

Plan

Perl et CGI

Rappel

Les CGI (common gateway interface), sont les programmes exécutés sur un serveur Web.
Ce sont des programmes qui s'exécutent sur le serveur Web et qui produisent (le plus souvent) de l'html.

Remarque

Un CGI est un programme (ici écrit en Perl) qui doit afficher un « header » (ici `Content-type : text/html` suivi d'une ligne vide), et qui ensuite affiche du html.

Perl et CGI

Premier CGI : sans paramètres

Exemple de petit programme CGI :
Il s'agit d'un programme qui affiche les fichiers d'un répertoire (commande UNIX `ls`).

```
#!/usr/bin/perl
print "Content-type: text/html\n\n"; # Header indispensable

# Corps du programme
open(LISTE, 'ls |')
    || die "Impossible d'exécuter la commande : $!";
print "Liste : \n<ul>";
while (my $fichier = <LISTE>) {
   print "<li>$fichier</li>\n";
}
print "</ul>\n";
```

Perl et CGI

Premier CGI : sans paramètres

Ce programme affiche un fichier html qui ressemble à :

```
Content-type: text/html
Liste :
<ul>
<li>essai.pl</li>
<li>essai2.pl</li>
...
<li>liste.pl</li>
</ul>
```

Perl et CGI

Deuxième CGI : associé à un formulaire

Le CGI précédent est un cas particulier relativement rare... En général un programme CGI est paramétrable. C'est-à-dire que l'utilisateur peut entrer des données qui seront analysées par le CGI. Ce qui, sous Web, sous-entend l'utilisation de formulaires.
Le cas général est donc : un formulaire pour entrer les données, et un CGI pour les exploiter. Rien n'oblige le formulaire à être sur le même serveur que le CGI. Par contre le CGI doit être dans un répertoire spécial du serveur Web (en général `/usr/lib/cgi-bin/programme.pl`) et sera accessible avec l'URL correspondante (en général `http ://serveur/cgi-bin/programme.pl`).

Perl et CGI

Deuxième CGI : associé à un formulaire

Exemple :

Un petit CGI poli, `bonjour.pl`

```
#!/usr/bin/perl
use CGI_Lite; # Utilisation d'un module CGI
my $cgi=new CGI_Lite; # Nouveau CGI
my %in = $cgi->parse_form_data; # Lecture des parametres
print "Content-type: text/html\n\n"; # Header indispensable
# Corps du programme
print "<h1>Bonjour $in{'nom'} !</h1>";
```

Ce CGI utilise un module spécial (`CGI_Lite`) qui permet de lire les données d'un formulaire et de les placer dans le tableau associatif `%in`.

Deuxième CGI : associé à un formulaire

Il reste ensuite à écrire un formulaire qui remplit les données dont on a besoin (pour l'exemple seul le paramètre `nom` est utilisé).

`bonjour.html` :

```
<form action="http://url/cgi-bin/bonjour.pl"
      method="POST">
   <b>Quel est votre nom ?</b>
   <input type="text" name="nom"/>
   <input type="submit" value="Envoyer"/>
</form>
```

Perl et CGI

Un exemple un peu plus complexe

Utilisation de différentes « balises » pour saisir des informations.
Un problème se pose avec les balises de type « sélection multiple » : nous avons plusieurs valeurs dans une même variable...
Pour résoudre ce problème le module CGI_Lite offre une fonction `$cgi->get_multiple_values ($in{'champ'})` qui permet de retourner sous forme de tableau les différentes valeur du champ.
On va créer un CGI qui crée une salade de fruits avec les paramètres de l'utilisateur.

Source du formulaire : salade.html

```
<form action="http:/url/cgi-bin/salade.pl"
      method="POST">
   <b>Quel est votre nom ?</b>
   <input type="text" name="nom"/><br/>
   <b>Que voulez-vous dans votre salade de fruit ?<br/>
   <input type="checkbox" name="fruit" value="amande"/>Amande<br/>
   <input type="checkbox" name="fruit" value="banane"/>Banane<br/>
   <input type="checkbox" name="fruit" value="cerise"/>Cerise<br/>
   <input type="checkbox" name="fruit" value="kiwi"/>Kiwi<br/>
   Et pour sucrer :
   <select name="sucre">
      <option checked="checked">sucre vanill&eacute;</option>
      <option>sucre de canne</option>
      <option>sirop</option>
      <option>rien du tout</option>
   </select><br/>
   Une remarque ?<br/>
   <textarea name="remarques" cols="40" rows="4"></textarea><br/>
   <input type="submit" value="Envoyer">
</form>
```

Le source de salade.pl est un petit peu plus complexe :

```
#!/usr/bin/perl
use CGI_Lite; # Module CGI_Lite...
my $cgi=new CGI_Lite; # On cree un objet de type CGI
my %in = $cgi->parse_form_data;# On lit les donnees
print "Content-type: text/html\n\n"; # Indispensable: Header
# Les champs du formulaire sont maintenant dans le tableau
# associatif %in. Le contenu d'un champ sera donc $in{'champ'}
# ATTENTION: Certains champs ont plusieurs valeurs
# dans ce cas on fait reference aux valeurs dans un tableau
# obtenu par $cgi->get_multiple_values ($in{'champ'})
print "Bonjour $in{'nom'} !<br/>\n";
print "Voici donc une petite salade de fruits :\n";
print "<br/>Compos&eacute;e de <ul>\n";
# Pour toute valeur
foreach my $f ($cgi->get_multiple_values ($in{'fruit'})) {
   print "<li>$f</li>\n";
}
print "</ul>Et pour la douceur un petit peu de $in{'sucre'}<br/>\n";
if (exists($in{'remarques'})) { # Si le champ a ete rempli...
   print "Nous avons tenu compte de votre remarque<br>\n";
   print "<b>$in{'remarques'}</b>\n";
}
```

Variante de print

```
print << "FinDeTexte"
tout
est
écrit
jusqu'ici
FinDeTexte
```

Plan

Accès aux bases de données : dbperl

Un des aspects les plus intéressants de Perl.
Il permet d'intégrer des requêtes SQL dans un programme.
Depuis Perl version 5, on accède de la même manière à une base de données quelque soit le système choisi.
Auparavant l'accès était différent si on utilisait Oraperl (Oracle), ou Syperl (Sybase)...
Maintenant on utilise un module DBI (database interface), et on spécifie à la connexion que l'on travaille sur une base Oracle, ou Ingres, ou Sybase...
Quand on installe Perl, il faut installer un module DBI.

Accès aux bases de données

Voici quelques commandes à utiliser pour accéder aux bases de données MySQL.

`use DBI ;` : en début de programme spécifie que l'on va faire de l'accès aux bases de données

`$db = DBI->connect("dbi :mysql :base", 'nom', 'mot-de-passe') ;` : connexion à MySQL, sur la base de données `base` avec le l'utilisateur `nom` et son mot de passe

`$db->disconnect ;` : pour se déconnecter de la base de données en fin de programme

Accès aux bases de données

`$db->do("requête") ;` pour exécuter une requête SQL.
À n'utiliser qu'avec des requêtes du genre `create table`, `update nom-table`, `insertinto nom-table`, ...
Il est préférable de traiter une erreur éventuelle dans ce genre d'opération (requête SQL mal formulée, connexion interdite, ...).
On le fait de la même manière que pour la détection des erreurs dans l'ouverture de fichier :

```
$db->do("requête") || die "Pb de requête : $DBI : :errstr" ;
```

(l'erreur est une erreur du SGBD, c'est pourquoi on utilise la variable `$DBI : :errstr`)
Pour une requête de type `select` on va procéder en 4 temps :

1. préparation de la requête (`prepare`)
2. exécution (`execute`)
3. parcours de chaque ligne retournée par la requête (`fetchrow_array`), dans une boucle
4. fin de la requête (`finish`)

Exemple : lister toutes les lignes d'une table

```
my $sel = $db->prepare("select * from table where condition");
$sel->execute || die "Pb de sélection : $DBI::errstr";
while (my ($champ1, $champ2, ...) = $sel->fetchrow_array) {
   print "Contenu: $champ1, $champ2, ... \n";
}
$sel->finish; # On ferme la requête select
```

Remarque : La lecture d'une ligne se fait donc avec `$sel->fetchrow_array`, qui retourne en fait un tableau (d'où son nom).
On pourrait donc écrire `@tab = $sel->fetchrow_array ;`
Il est possible de paramétrer une requête, en mettant des points d'interrogation dans la requête au niveau du `prepare`, les paramètres seront spécifiés dans le `execute`.

Exemple :
Afficher le nom d'un patient dont le numéro est demandé à l'utilisateur

```
my $sel = $dbh->prepare("select prenom from personne
                         where nom = ?");
print "Veuillez entrer un nom : ";
my $nom = <STDIN>; # lecture au clavier (entrée standard)
chomp($nom); # ne pas oublier d'enlever le retour-chariot
$sel->execute($nom) || die "Pb de sélection : $DBI::errstr";
while (my ($prenom) = $sel->fetchrow_array) {
   print "Prénom : $prenom \n";
}
$sel->finish;
```

Voici maintenant un exemple de programme Perl qui crée une table `patient` avec trois champs (`numero`, `nom`, `prenom`), et qui demande de saisir une liste de noms - prénoms, et qui, pour chaque ligne, insère les données dans la table (le programme se charge d'incrémenter à chaque fois le numéro de patient) :

```
#!/usr/bin/perl
use strict;
use DBI; # On va utiliser la base de données
# Connexion à la base de données
my $db = DBI->connect("dbi:mysql:test", 'root','root');
# Création de la table patient :
$db->do("create table patient (
numero number(10) not null primary key,
nom    varchar2(40),
prenom varchar2(20))") || die "Pb création table: $DBI::errstr";
# Préparation d'une requête d'insertion des valeurs dans la BDD
my $ins = $db->prepare("insert into mpatient values (?, ?, ?)");
my $nopatient=0; # Initialisation du compteur
print "Entrez une série de nom-prenom, finir par CONTROL-D\n";
while (my $ligne = <>) { # Pour chaque ligne lue...
   # Séparation en nom,prenom
   # caractère de séparation: tabulation
   my ($nom, $prenom) = split(/\t/, $ligne);
   # Exécution de la requete (insert), détection de l'erreur
   $ins->execute($nopatient, $nom, $prenom) || die "Pb à l'insertion : $DBI::errstr
   $nopatient++; # Incrémentation du compteur
}
$ins->finish; # On ferme la requête insert
$db->disconnect; # Déconnexion de la BDD
```

Voici maintenant un programme qui affiche le contenu de la table `patient` (que l'on vient de créer).

```
#!/usr/bin/perl
use strict;
use DBI; # On va utiliser la base de données
my $db = DBI->connect('dbi:mysql:test', 'root','root');
# Préparation d'une requête de sélection des valeurs
my $sel = $db->prepare('select * from patient');
$sel->execute || die "Pb à la sélection : $DBI::errstr";
print "Voici la liste des patients enregistrés";
while (my ($nopatient, $nom, $prenom) = $sel->fetchrow_array)
   # Pour chaque ligne lue...
   print "Patient no $nopatient : $nom, $prenom\n";
}
$sel->finish; # On ferme la requête insert
$db->disconnect; # Déconnexion de la BDD
```

Accès à une base de données depuis le web

Pour accéder à une base de données depuis le web, cela sous-entend que l'on va utiliser des programmes CGI qui accèdent à une base de données (CGI et DBperl).
Voici un petit exemple simple qui liste tous les patients de notre base de données sous la forme d'un tableau (on demandera au préalable de saisir un motif de recherche des patients).

```
<form action="http://localhost/cgi-bin/listepatient.pl"
      method="POST">
   <b>Votre recherche ? </b>
   <input type="text" name="recherche"/>
   <input type="submit" value="Envoyer"/>
</form>
```

le programme listepatient.pl

```
#!/usr/bin/perl
use strict;
use DBI; # On va utiliser la base de donnees
use CGI_Lite; # Module CGI_Lite...
my $cgi=new CGI_Lite; # On cree un objet de type CGI
my %in = $cgi->parse_form_data; # On lit les donnees
print "Content-type: text/html\n\n"; # Indispensable: Header
my $db = DBI->connect('dbi:mysql:test','root','root');
# Preparation d'une requete de selection des valeurs
my $sel=$db->prepare('select * from patient where nom like ?');
my $recherche = "%$in{'recherche'}%";
$sel->execute($recherche) || die "Pb à la sélection : $DBI::errstr";
print "Patients dont le nom contient $in{'recherche'}\n";
print "<table border='1'>\n";
print "<tr><th>Num&eacute;ro</th><th>nom</th><th>pr&eacute;nom</th></tr>\n";
my $nblignes=0;
while (my ($nopat, $nom, $prenom) = $sel->fetchrow_array) {
   # Pour chaque ligne lue...
   print "<tr><td>$nopat</td><td>$nom</td><td>$prenom</td></tr>\n";
   $nblignes++;
}
print "</table>\n";
print "Nombre de patients : $nblignes";
$sel->finish; # On ferme la requête insert
$db->disconnect; # Déconnexion de la BDD
```

Références bibliographiques

J. Tisdall.
Introduction à Perl pour les biologistes.
O'Reilly, 2002.
Traduit par L. Mouchard et G. Ricard.

B. Pouliquen.
Introduction au langage Perl.
http://www.med.univ-rennes1.fr/~poulique/cours/perl