Initiation au langage PERL

par Philippe Bousquet <Darken33@free.fr>

Initiation au langage PERL

par Philippe Bousquet <Darken33@free.fr>

Ce livre électronique est une réédition d'un article de préparation d'une conférence que j'avais effectué au lycé Casler à Talence en février 2003 dans le cadre des « conférences de l'ABUL ».

Ce livre a été écrit à partir des travaux d'Alain FORCIOLI (aforcioli@april.org), membre de l'APRIL, sur "Le Langage Perl".

Table des matières

Introduction

Qu'est ce que le PERL ?

P.E.R.L signifie "Practical Extraction and Report Language" que l'on pourrait traduire par "Langage Pratique d'Extraction et d'Edition". Il a été créé par Larry Walls en 1986, à l'origine pour gérer un système de "News" entre deux réseaux.

Il s'agit d'un langage interprété, c'est à dire qu'il n'est pas nécessaire de compiler un programme (comme en C) pour pouvoir l'exécuter. Le PERL combine plusieurs des meilleures fonctions du langage C et des Shell Unix, il possède également l'ensemble des outils annexes permettant de traiter des chaînes de caractères tels que sed, awk ou tr.

Le succès du PERL est notamment due :

- Au fait qu'il soit gratuit, tout en comprenant un grand nombre de modules
- A sa portabilité : En effet il existe des versions de PERL pour à peu près tous les systèmes d'exploitations (Linux, Unix, Windows, Mac, Amiga, Atari, ...)
- A sa simplicité : Avec quelques lignes de commandes on peut faire ce que fait un programme C ou Pascal en 500 lignes.
- A sa robustesse : Il n'y a en effet pas d'allocation mémoire à gérer, les chaînes, piles noms de variables

Le PERL : Pour quelle utilisation ?

A l'origine le 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)

Le langage a depuis évoulé pour aujourd'hui être surtout utilisé pour :

1. générer, mettre à jour et analyser des fichiers HTML (notamment à travers des scripts CGI)
2. accéder aux bases de données
3. faire des conversions de format de fichiers

Cependant le Perl n'est pas prévu pour :

1. la réalisation d'interfaces graphiques (bien qu'il existe le module perl/tk permettant cette fonctionnalité)
2. le calcul scientifique (pour un problème de performances)

Aperçu de programmes PERL

Mon premier programme en PERL (bonjour.pl) :

```
#!/usr/bin/perl
# Affiche Bonjour à l'écran
print "Bonjour !!!\n";
```

Un programme PERL commence obligatoirement par la ligne de commentaire `"#!/.../perl"` où l'on spécifie le chemin de l'interpréteur perl.

Ce petit programme permet d'afficher « Bonjour !!! » à l'écran.

Affichage du nombre de ligne d'un fichier (nblignes.pl) :

```
#!/usr/bin/perl
# Ouverture du fichier test.txt en lecture
# Si une erreur survient afficher un message et sortir
open (F, "$ARGV[0]") || die "Problème d'ouverture : $!.\n";
# Lecture globale du fichier
@lignes=<F>;
# Fermeture du fichier
close F;
# Recuperation du nombre de lignes du fichier
$nblg=$#lignes+1;
# Affichage du resultat
print "le fichier $ARGV[0] contient $nblg lignes.\n";
```

Ceci est un petit exemple de la puissance du langage PERL.

En effet ici nous n'avons pas besoin de gérer de boucle de lecture et donc nous arrivons à écrire un programme permettant de compter le nombre de lignes d'un fichier avec simplement six lignes de code. Dans un langage tel que le C, il en faudrait nettement plus.

Exécuter un programme PERL :

Il y à deux possibilité pour exécuter un programme PERL, la première etant d'appeler l'interpréteur en tapant la commande suivante :

```
$ perl bonjour.pl
Bonjour !!!
$
```

La seconde consiste a rendre le programme executable en changeant ses droits :

```
$ chmod 755 nblignes.pl
$ ./nblignes.pl bonjour.pl
le fichier bonjour.pl contient 3 lignes.
$
```

Les types de données

Le PERL permet de gérer les types de données suivants :

- Tableau : Liste indexable de valeurs scalaires.
- Code : Un bout de code Perl, par exemple un sous-programme.
- Handle de Fichier : Utilisés dans les opérations d'entrées et de sorties.
- Hachage Tableau Associatif de valeurs scalaires.
- Scalaire : Chaînes, nombres et références.

Variable Scalaire

Description

C'est la variable de base. La variable scalaire se préfixe du `$` (comme les variables shell). Elle peut prendre pour valeur un entier, une chaîne de caractères ou, une référence.

Remarque: dans un code PERL, le caractère `#` permet de commenter tout ce qui suit jusqu'au retour chariot.

```
$number = 1;               # un entier
$reel = 1.5                # un nombre flottant
$string = "Il fait beau"; # une chaîne de caractères
$reference = \$number;     # une référence sur l'entier 1
```

Exemples d'opérations sur les variables scalaires

```
$i = "1";  # la chaine "1"
$i++;      # l'entier 2
$i = "$i"; # la chaine "2"
```

PERL effectue les conversions implicites des valeurs des variables scalaires pour que les opérations qui les affectent puissent être exécutées.

```
$phrase = "2+2 = 4 " . "et sqrt(4) = 2" . ".\n";
```

Le caractère `.` permet de concaténer des chaînes entre elles.

```
$phrase = "2+2 = ", 2+2, " et sqrt(4) = ", sqrt (4), ".\n";
```

La virgule permet de concaténer les évaluations des routines présentes sur la ligne d'affectation. Ainsi la variable scalaire `$phrase` contient la chaine `"2+2 = 4 et sqrt(4) = 2."`.

La table ou liste

Description

Préfixé du caractère `@`, le tableau appelé aussi table ou liste est un ensemble de variables scalaires. Donc une table peut contenir des entiers, des chaînes de caractères, des références et même des tables. Pour définir une table on utilise les parenthèses et on sépare les éléments par une virgule.

```
(a, b, c)             # une table de caractères
@tab = ( 1, 2, 3)     # une table d'entiers
@foo = ( 1, b, $var) # une table avec des éléments de
                      # types différents
```

Une table est indexée par des entiers. Le premier indice d'une table est 0. Le premier élément de la table `@tab` se nomme `$tab[0]`. L'indice du dernier élément de la table est `$#tab`. Le nombre d'éléments de la table peut donc s'écrire `$#tab + 1`

```
@tab = (a, b, c);
$tab[0];           # a
$tab[1];           # b
$#tab;             # 2
$tab[$#tab];       # c
$#tab + 1;         # 3
```

Les fonctions utilisant les tables

(Se reporter à la section *perlfunc* du manuel Perl.)

push, pop, shift, unshift

Ces fonctions permettent de manipuler la table comme une pile. La fonction `pop` retourne le dernier élément de la table et le supprime. `push` ajoute un élément à la fin d'une table. `shift` retourne le premier élément d'une table et le supprime. `unshift` insere un élément au début d'une table.

Voici quelques exemples d'utilisation de ces fonctions.

```
@tab = ('Larry', 'WALL');
$age = '24';
push(@tab, $age);     # @tab = ('Larry', 'WALL', '24')
print shift(@tab);    # affiche 'Larry' et @tab = ('WALL', '24')
print pop(@tab);      # affiche '24' et @tab = ('WALL')
```

sort, reverse, split, join

La commande `sort` trie les éléments d'une table et retourne la liste triée.

```
@tab = ('b', 'w', 'a', 't');
@tab = sort @tab;              # @tab = ('a', 'b', 't', 'w')
```

La commande `reverse` retourne la liste des éléments dans l'ordre inversé.

```
@tab = reverse @tab;           # @tab = ('w', 't', 'b', 'a')
```

`split` permet de découper une chaîne de caractères. C'est une fonction puissante car c'est l'utilisateur qui spécifie quel caractère ou quelle expression régulière (voir chapitre sur les expressions régulières) permet de délimiter les éléments à découper. La commande retourne une liste contenant les éléments de la chaîne découpée.

```
@tab = split(/\+/, "1 + 2 + 3");     # @tab vaut (1, 2, 3)
```

Ici le caractère de séparation est `+`. Si on avait voulu découper une chaîne contenant des `+` et des `-` on aurait utilisé l'expression régulière suivante : `/\+|-/` qui veut dire "+ ou -".

Remarque : le caractère `+` est précédé de `\` car s'est aussi un mot clé pour les expressions régulières, mais nous verrons ça plus tard.

`join` regroupe les éléments d'une table dans une chaîne de caractères en les séparant par la chaîne de caractères donnée en argument. La chaîne de caractères peut être annotée comme une expression régulière ou entre guillements.

```
print join(/ - /, @tab);        # affiche '1 - 2 - 3';
print join(" - ", @tab);        # idem
```

La table de hash

Introduction

Une fonction de hash permet de calculer une clé (unique si possible) à partir d'une chaîne de caractères. La clé de valeur entière permet d'indexer une table par la suite. Cette technique est appelée *Hash coding*. Son avantage est qu'elle diminue considérablement le nombre de tentatives lors d'une recherche dans une table.

Une table de hash est une liste dont chaque élément est un couple de la forme `(nom, valeur)`. *nom* est une chaîne de caractères (ex: "123", "coucou", etc...) dont PERL extrait (par une fonction de hash) une clé. Cette fonction est invisible pour le programmeur.

Donc à l'indice correspondant à la clé de `nom` on trouve l'élément `valeur`. Une table de hash est en général utilisée pour construire une structure de données. Dorénavant, et par abus de langage, je dirais que `nom` est la clé.

Remarque: awk (gawk) fournit également ce mécanisme.

Description

La forme générale d'une table de hash est :

```
%hash = ( cle1, val1, cle2, val2, ..., cleN, valN);
%hash = (nom, WALL, prenom, Larry);
```

L'exemple montre une table avec deux éléments. La valeur associée à la clé `prenom` est `Larry`. De même la valeur associée à la clé `nom` est `WALL`.

Cette notation est maintenant obsolète et peu lisible. Une nouvelle écriture permet de ne pas confondre la table de hash avec une liste.

```
%hash = ('nom' => 'WALL', 'prenom' => 'Larry');
$hash{'nom'};                                   # WALL
$hash{nom};                                     # WALL
$hash{prenom};                                  # Larry
```

On remarque que lorsque la clé est constituée d'un seul mot il n'est pas nécessaire de la mettre entre apostrophes (ou *quotes*).

Fonctions usuelles

Les exemples des fonctions que nous allons décrire seront basés sur la table de hash suivante:

```
%hash = (
 'nom' => 'BOUSQUET',
 'prenom' => 'Philippe',
 'age' => '26'
);
```

keys, values

`keys` retourne une liste (table) contenant les clés de la table de hash.

```
@k = keys %hash;          # @k = ('nom', 'prenom', 'age')
```

Sachant que keys retourne une liste, il est facile de l'avoir de manière ordonnée avec la fonction `sort`.

```
@k = sort keys %hash;     # @k = ('age', 'nom', 'prenom')
```

`values` à l'inverse de `keys` retourne une liste de toutes les valeurs de la table. L'obtention d'une telle liste ordonnée se fait de cette manière :

```
@v = sort values %hash;  # @v=('26', 'BOUSQUET', 'Philippe')
```

each, delete, exists

`each` retourne le premier couple de la table de hash sous la forme d'une liste. Le couple est ensuite supprimé de la table de hash.

```
($nom, $val) = each %hash; # $nom = 'nom'
                           # $val = 'BOUSQUET'
                           # keys %hash = ('prenom', 'age')
```

`delete` supprime la valeur associée à la clé donnée en argument.

```
delete $hash{'age'};       # keys %hash = ('prenom')
```

`exists` retourne 1 (vrai) si une valeur existe pour une clé donnée.

```
print "Exists\n" if exists $hash{'prenom'}; # affiche 'Exists\n'
```

Les Variables et Tableaux Spéciaux

Les Variables Spéciales :

$_	La dernière ligne lue (au sein d'une boucle while)
$!	La dernière erreur (utilisé dans le détection d'erreur)
$$	Le numéro de processus du programme
$1,$2,...	le condenu de la parenthèse N dans la dernière expression régulière utilisée
$0	Le nom du programme (peut servir pour gérer plusieurs entrées)

Les Tableaux Spéciaux :

@_	Contient les paramètres passés à une fonction
@ARGV	Contient les paramètres passés au programme
%ENV	Tableau indicé contenant l'ensemble des variables d'environnement
@INC	Contient l'ensemble des des repertoires contenant des bibliothèques

Syntaxe générale

Chaque instruction doit être terminée par un point-virgule. Un passage à la ligne ne signifie pas une fin d'instruction (ce qui est souvent une source d'erreurs au début).

```
# ce programme est érroné !
$a = 2
print $a;

# Par contre ceci est tout a fait correct
$a = "ceci est un longue chaîne sur plusieurs
lignes...";
```

Les commentaires commencent par le carractère #
Tout le reste de la ligne est alors considérée comme un commentaire

```
# ceci est un commentaire !
$a = 2    # et en voici un autre
```

Un bloc est un ensemble de commandes entourées par des crochets (`{}`), chaque commande étant suivie d'un point-virgule

Les Opérateurs

Opérateur	Description
`->`	Opérateur de déréférence (infix)
`++`	Autoincrémentation
`--`	Autodécrementation
`**`	Exponentiation
`\`	Référence à un objet
`! ~`	Négation, complément
`=~`	Lie une expression scalaire à un motif
`!~`	Même chose mais retourne la négation du résultat
`* / % X`	Multiplication, Division, Modulo, Répétition
`+ - .`	Addition, Soustraction, Concaténation
`>> <<`	Décalage binaire à droite, à gauche
`< > <= >=`	Opérateurs relationnels numériques
`lt gt le ge`	Opérateurs relationnels de chaînes
`== != <=>`	Egal, non egal, comparasion numériques
`eq ne cmp`	Egal, non égal, comparaison de chaînes
	Les comparaisons retourne (-1 : plus petit, 0 : égal, 1 : plus grand)
`&`	Et binaire
`| ^`	Ou binaire, OU binaire exclusif
`&&`	Et logique
`||`	Ou logique
`..`	Intervalle
`?:`	Opérateur conditionnel (*si ? alors : sinon*)
`= += -= etc.`	Opérateur d'affectation
`,`	Opérateur virgule, également séparteur de liste
`=>`	Référence de hash
`not`	Non logique
`and`	Et logique
`or`	Ou logique
`xor`	Ou logique exclusif

Les Expressions Conditionelles

Condition : if, elsif, else

L'instruction `if` prend la syntaxe suivante :

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

PERL va évaluer la *condition*, si celle ci est vrai alors le *bloc* sera exécuté.

```
if ($prix > 200) {
  print 'ceci est un peu cher...';
  print 'il faut négocier...';
}
```

On peut vouloir exécuter un autre bloc si la condition n'est pas vrai, ceci est faisable avec le mot clé `else` :

```
if (condition) {
  bloc1;
}
else {
  bloc2;
}
```

Si la *condition* est vrai, PERL executera le *bloc1*, sinon il exécutera le *bloc2*.

```
if (($nombre % 2) == 0) {
  print "$nombre est pair.\n";
}
else {
  print "$nombre est impair.\n";
}
```

On peut vouloir effectuer plusieurs cas de conditions, avec des traitements particuliers pour chacune on peut utiliser pour cela l'instruction `elseif` :

```
if (condition1) {
   bloc1;
}
elsif (condition2) {
   bloc2;
}
```

Si *condition1* est évaluée comme vrai alors *bloc1* sera exécuté, Sinon si *condition2* est évaluée comme vrai alors *bloc2* sera. Exécuté

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

Il existe une notation un peu particulière d'un conditionnement en PERL qui ne fonctionne que pour une seule instruction à exécuter lorsque la condition est vrai :

```
instruction if (condition);
```

Instruction sera exécuté si *condition* est vrai.

```
print "J'achète..." if ($prix <= 20);
```

Condition inverse : unless

L'instruction unlesse permet d'exécuter un bloc, si la condition est évaluée à faux.

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

Si *condition* est évaluée comme fausse alors *bloc* sera exécuté.

```
unless ($prix <= 200) {
  print 'ceci est un peu cher....';
  print 'il faut négocier...';
}
```

Comme pour le if il existe un codification un peut particulière qui fonctionne pour une seule instruction à éxecuter :

```
commande unless (condition);
```

Les Boucles

Tant que : while

La boucle `while` pemet d'executer un bloc d'instructions tant qu'une condition est vrai.

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

tant que *condition* est vrai PERL exécute *bloc*

```
$mon_argent = 10;
while ($mon_argent >= $prix('cerise')) {
  $mon_argent -= $prix('cerise');
  print 'Et un kilo de cerises !';
}
```

il existe ici aussi une codification particulière que l'on peut utiliser si l'on doit n'executer qu'une instrauction dans la boucle.

```
instruction while (condition);
```

instruction sera exécutée tant que *condition* sera vrai

Répétititon : do ... while

Le couple `do ... while` fonctionne à peu près comme la boucle while, la se »ule différence se situant sur le fait que l'on exécute une première fois le bloc.

```
do {
   bloc;
}
while (condition);
```

exécute au moins une fois *bloc*, puis l'exécute tant que *condition* est vrai

```
# recherche le premier fruit <= 2 Euros
$i=0;
do {
  $f = $fruit[$i];
  $i++;
}
while ($prix($f)>2)
```

Le couple `do ... until` permet quant à lui d'exécuter un bloc au moins une fois ju'squ'a ce q'une condition devienne vrai.

```
do {
   bloc;
}
until (condition);
```

execute au moins une fois *bloc*, puis l'execute jusqu'à ce que *condition* soit vrai

```
# recherche le premier fruit <= 2 Euros
$i=0;
do {
  $f = $fruit[$i];
  $i++;
} until ($prix($f)<=2)
```

La boucle Pour : for

Cette boucle, permet de positionner une initialisation en debut de boucle ainsi que d'exécuter une instruction à chaque itérration.

```
for (init;condition;instruction) {
   bloc;
}
```

L'utilisation principale de cette boucle est l'execution d'un bloc un nombre fini de fois. Par exemple pour le parcours d'un tableau

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

Pour chaque élément : foreach

Cette boucle est une simplification de la boucle `for`, en effet elle s'applicque à tous les élements d'un tableau.

```
foreach élément (tableau) {
   bloc;
}
```

exécute *bloc* pour chaque *élément* de *tableau.*

```
# Cette méthode est moins verbeuse que la boucle for
foreach $f (@fruit) {
  print "$f\n";
}
```

Les Fonctions

Déclaration

La plupart du temps, vous aurez à utiliser le même bout de code à plusieurs endroit de votre programme, PERL permet donc de déclarer des routines pouvant être appelées à n'importe quel moment dans le code d'un programme.

```
sub ma_procedure {
  bloc;
}
```

pour appeler une fonction dans un programme vous pouvez utiliser la notation suivante `&ma_procedure();` ou plus simplement `ma_procedure;`.

Je vous conseille cependant de préférer la première notation pour une simple question de lisibilité.

Avec paramètre(s)

L'intéret principal d'extraire un bout de code dans une fonction est de pouvoir y transmètre des paramètres. PERL permet bien entendu cela :

```
sub ma_fonction {
 my ($var1, $var2, ...) = @_;
 bloc;
}
```

Il suffit allors d'appeler la fonction dan son programme de la façon suivante :

```
&ma_fonction(valeur1, valeur2, ...);
```

Par exemple une fonction de calcul du carré d'une valeur :

```
sub carre {
  my ($valeur) = @_;
  print "$valeur au carré = ",($valeur*$valeur),".\n";
}

# partie principale
print "Bonjour, voici un petit programme de calcul\n";
&carre(2);
# résultat => "2 au carré = 4"
&carre(3);
# résultat "3 au carré = 9"
```

Remarque :

L'instruction `my` réalise une affectation dans des variables locales à la procédure avec les éléments du tableau. Les paramètres sont passés dans le tableau spécial `@_`, on peut éventuellement référencer chaque paramètre par cahque élément du tableau :`$_[0]`, `$_[1]`, ... souvent au détriment de la lisibilité.

```
sub carre2 {
  print $_[0]." au carré = ",($_[0]*$_[0]),".\n";
}

# partie principale
print "Bonjour, voici un petit programme de calcul\n";
&carre2(2);
# résultat => "2 au carré = 4"
&carre2(3);
# résultat "3 au carré = 9"
```

Fonctions retournant un résultat

Encore une fonctionnalité interressante pour une fonction est de pouvoir retourner un résultat à l'appelant, ceci se fait via le mot clé `return` :

```
sub carre3 {
  my ($valeur) = @_;
  $resultat=$valeur*$valeur;
  return ($resultat);
}

# partie principale
print "Bonjour, voici un petit programme de calcul\n";
$valeur=&carre3(2);
print "2 au carré = ".$valeur."\n";
# résultat => "2 au carré = 4"
print "3 au carré = ",&carre3(3),"\n";
# résultat "3 au carré = 9"
```

Modules

La plupart du temps vous aurez besoin d'utiliser des fonctions définis dans des fichiers extérieurs ou dans des modules, pour les importer trois fonctions sont disponibles :

Syntaxe	Description	Exemple
import *module;*	Importe des fonctions et des variables de modules (sans vérifier l'existence de celui ci)	`import CGI;`
use *module;*	idem que import mais vérifie l'existence du module, si celui ci n'existe pas cela provoque un abort.	`use CGI;`
require *fichier;*	Permet d'inclure un fichier perl, si celui ci n'est pas trouvé cela provoque un abort	`require find.pl;`

La Gestion de Fichiers

Comme le langage PERL a été développé pour pourvoir traiter de manière simple les fichiers, il est normal qu'il possède quantité de fonctions relatives au traitement de fichiers. Seules les fonctions de base seront abordées ici.

En PERL, un fichier peut être un fichier physique situé sur un disque, mais aussi le clavier (STDIN), l'écran (STDOUT) et la sortie erreurs (STDERR).
L'entrée STDIN est souvent employée quand le script PERL doit traiter des informations en provenance d'un formulaire, puisque les données y sont stockées.

Ouverture

L'ouverture d'un fichier se fait via la commande :

```
open (handle, "nom_fichier");
```

handle est l'identifiant du fichier il est de bon ton qu'il soit en majuscule.
nom_fichier contient le chemin d'acccès au fichier (par exemple /home/darken/fichier.txt).
On ajoute aussi généralement avant le nom du fichier, la manière dont on va accéder au fichier.

Commande	Description
`open(nom_symbolique," nom_fichier")`	Ouvre le fichier en lecture
`open(nom_symbolique, "<nom_fichier")`	L'ouvre aussi en lecture
`open(nom_symbolique, ">nom_fichier")`	Ouvre le fichier en écriture. S'il n'existe pas il est créé.
`open(nom_symbolique," >>nom_fichier")`	Ouvre le fichier en ajout, tout ce qui est écrit dans le fichier sera placé à la fin de celui-ci.
`open(nom_symbolique, "+>nom_fichier")`	Ouvre le fichier en lecture et en écriture
`open(nom_symbolique, | PROGRAM)`	Envoie les données imprimées dans le fichier nom_symbolique vers le programme PROGRAM
`open(nom_symbolique, PROGRAM |)`	Utilise les données envoyées par PROGRAM comme source de données.

Lecture

Plusieurs méthodes existe pour lire un fichier, la plus utilisé est :

```
$ligne = <handle>;
```

Ceci permet de lire une ligne du fichier handle et de mettre son contenu dans la variable `$ligne`.

On peut également pour des fichiers pas trop imposant effectuer la lecture globale du fichier en une seule instruction :

```
@lignes = <handle>;
```

Dans ce cas, le contenu du fichier se trouve dans le tableau (une ligne par élément du tableau).

Enfin on peut lire une partie du contenu d'un fichier par l'instruction :

```
read(handle, $buffer, longeur, position);
```

Cette instruction va lire `longueur` caractères à partir de la `position` dans le fichier `handle`, le résultat sera mis dans la variable `$buffer`.

Ecriture

L'écriture dans un fichier peut se faire de manière relativement simple en utilisant l'instruction :

```
print handle texte;
```

Cela imprime dans le fichier `handle` les données contenues dans `texte`.
Quand on ne spécifie pas de nom de fichier, cela signifie que l'on affiche `texte` à l'écran.

`texte` peut contenir des constantes mais aussi des appels de variables comme dans l'exemple suivant :

```
$nom = 'Philippe';
print "Bonjour, mon nom est : $nom\n";
# va afficher à l'écran => Bonjour, mon nom est : Philippe
```

Une autre instruction de sortie est l'instruction printf. Tout comme en C, elle permet de spécifier le format de sortie des données.
Sa syntaxe générale est :

```
printf handle (format,$variable);
```

Cela permet d'aligner des nombres sur le signe décimal comme le montre l'exemple suivant :

```
$CoutJanvier = 123.34;
$CoutFevrier = 23345.45;
printf("Janvier = \$%8.2f\n",$CoutJanvier);
printf("Fevrier = \$%8.2f\n",$CoutFevrier);
# Janvier = $  123.34
# Fevrier = $23345.45
```

Le PERL possède également toutes sortes de fonctions permettant de se positionner dans un fichier à un endroit donné, de supprimer des fichiers, de bloquer l'accès à des fichiers (sémaphores pour les accès concurrents sous Unix uniquement).

Fermeture

La fermeture d'un fichier se fait via la commande :

```
close(handle);
```

Le fichier spécial <>

Perl offre une fonctionnalité bien pratique : l'utilisation du fichier spécial `<>` qui permet de lire le contenu de l'entrée standard.

```
#!/usr/bin/perl
$nblg=0
while (<>) {
  $nblg++;
}
print "Le nombre de lignes lues : " . $nblg . "\n";
```

Ce programme permet de compter le nombre de lignes qui seront entrées sur l'entrée standard.

Les Expressions régulières

(Voir la section *perlre* du manuel de Perl)

5.1 Description

Les expressions régulières sont des caractères permettant de coder n'importe quelle entité écrite appartenant à un ensemble (ex: les décimales, les majuscules, les mots commencant par *z* ou une minuscule, etc...).

Elles sont largement utilisées par les développeurs UNIX et sont implémentées dans des commandes telles queawk, `sed`, `ed`, `vi`. Même le langage C dispose d'une librairie d'expressions régulières.

Le principe de ces caractères est de construire un masque codant l'entité cible. Ensuite chaque séquence de caractères est comparée au masque. S'il y a correspondance, la séquence appartient à l'ensemble de l'entité.

Ce mécanisme permet de faire des remplacements de chaînes de caractères dans un fichier sans savoir ou sont placées les chaînes à remplacer. Il permet d'extraire des informations sans en connaître la position dans un ou plusieurs fichiers. C'est de loin le meilleur outil d'extraction et de traitement d'informations dans un fichier et c'est pourquoi Perl l'utilise.

Illustrons l'usage des expressions en codant l'ensemble de tous les mots à caractères minuscules.

```
[a-z]
```

`[ ]` permet de définir un ensemble de caractères. Le caractère `-` signifie que l'ensemble contient la lettre a, b, c, ... ou z. On aurait pu écrire:

```
[abcdefghijklmnopqrstuvwxyz]
```

`philippe` appartient appartient à l'ensemble des mots composés de lettres minuscules. Par contre `Philippe` non, car le mot contient la majuscule P.

De manière générale une E.R. est préfixée et suffixée du caractère `/`.

```
/[a-z]/
```

Voici une rapide description de quelques caractères d'expression régulières.

Code	Description
`.`	n'importe quel caractère alphanumérique et d'espacement.
`+`	fait correspondre au moins 1 fois ou plusieurs ce qui précède
`?`	0 ou au plus 1 caractère de ce qui précède => a? permet de faire correspondre `''` ou `'a'`
`*`	0 ou plusieurs caractères de ce qui précède => a* peut faire correspondre `''`, `'a'`, `'aa'`, `'aaa'`, etc...
`[]`	un intervalle
`[^]`	un complément
`^`	qui commence par
`$`	qui finit par
`()`	Définit un groupe de caractères réutilisable
`|`	ou

Perl étend le jeu de caractères des expressions.

Code	Description
`\s`	n'importe quel caractère d'espacements
`\S`	le complément de \s

Perl permet de modifier le traitement des chaînes par des options, en passant une ou plusieurs lettres après le/ final de l'E.R :

Code	Description
`i`	les majuscules sont traitées comme des minuscules
`m`	traite la chaine de caractères commes si elle était constituée de plusieurs lignes
`s`	traite la chaine comme si elle était sur une seule ligne

Deux opérations sont principalement utilisées par les expressions régulières. La substitution et le *pattern matching* (la mise en correspondance).

```
s/source/target/options
```

`s` signifie substitution. `/` est le séparateur d'expression. Si une chaîne lue correspond à `source` elle est remplacée par `target`. `options` agit sur le nombre de substitutions. Si `g` est utilisé alors toutes les chaînes de la ligne lue correspondant à `source` sont remplacées (`g` pour global). `e` permet d'évaluer `target`.

Ainsi pour convertir un fichier texte en un fichier ne comportant que des majuscules on aurait le code suivant (fichier `uc.pl`):

```
#! /usr/bin/perl

while(<>) {
  s/^(.*)$/uc($1)/e;
  print $_;
}
```

On peut tester cette opération de la façon suivante:

```
$ uc.pl /etc/passwd
```

Le fichier `/etc/passwd` s'affiche avec uniquement des caractères majuscules.

5.2 Exemple: wc.pl

Le programme `wc.pl` suivant effectue le meme travail que la commande unix, à savoir comptabiliser le nombre de caractères, de lignes et de mots d'un fichier donné en argument (fichier `wc.pl`):

```
#! /usr/bin/perl

# Retourne le nombre de mots dans une ligne par mot on comprendra
# tous ce qui n'est pas 'espace'.
sub compte_mot {
  my($ligne) = @_;
  $ligne =~ s/\s+/" "/g;
  my(@tab) = split( /" "/, $ligne);
  $#tab;
}

sub compte_ligne {
  my($ligne) = @_;
  return 1 if $ligne !~ /^\s*$/;
}

sub compte_char {
  my($ligne) = @_;
  length($ligne);
}

sub main {
  my($mots, $lignes, $chars);
  while(<>) {
    $mots += compte_mot($_);
    $chars += compte_char($_);
    $lignes ++ if compte_ligne($_);
  }
  print "w = $mots, l = $lignes, c = $chars\n";
}

main;
```

Les références

Description

La référence Perl peut être vue comme le pointeur du langage C. C'est une variable qui en référence une autre. Pour annoter la référence sur la variable scalaire contenant la valeur 1, nous écrirons:

```
$i = 1;                # i est une var. scalaire et vaut 1
$scalar_ref = \$i;     # scalar_ref référence i.
```

Pour accéder au contenu pointé par la référence, il faut la déréférencer. Perl utilise l'opérateur `->` pour déréférencer une référence.

Pour accéder à la valeur de la variable pointée il faut aussi la déréférencer. De même qu'avec le langage C, une référence peut *pointer* une autre référence et ainsi de suite. Il faudra donc déréférencer autant de fois que nécéssaire pour obtenir la valeur.

Le valeur pointée peut être modifiée en manipulant la référence.

```
#!/usr/bin/perl
$i = 3;
printf("i = %d\n", $i);
$ref_i = \$i;
$$ref_i ++;
print $i, "\n";
```

Référence sur un tableau

La référence d'une table peut être créée de plusieurs façons:

```
@tab = (1, 2, 3);
$ref = \@tab;
```

ou

```
$ref = [1, 2, 3];
```

Pour déréférencer la table, il faut la préfixer du caractère qui désigne la table en Perl : @

```
@tab1 = @$ref;
```

Pour parcourir les éléments de la table on peut écrire:

```
foreach (@$ref) {
  # $_ contient successivement 1, 2 et 3
}
```

On peut directement accéder à un élément contenu dans la table pointée.

```
$ref->[0]; # vaut 1
$ref->[1]; # vaut 2
```

Pour obtenir le nombre d'éléments pointés :

```
$#{@$ref};
 | -----
 |    |
 |    retourne la table et non une référence
 |    (1, 2, 3)
 |
 |
 permet d'obtenir le nombre d'éléments d'une table.
```

L'expression précédente est équivalente à :

```
$#(1, 2, 3);
```

Référence sur une table de hash

De même qu'avec la table il existe divers moyens de créer une référence sur une table de hash.

```
%hash = (
          'un' => 1,
          'deux' => 2,
          'trois' => 3
        );

$ref = \%hash;
```

ou

```
$ref = {
         'un' => 1,
         'deux' => 2,
         'trois' => 3
       };
```

Pour déréférencer la référence il faut utiliser le caractère qui désigne la table de hash : `%`.

```
%hash = %$ref;
```

Le parcours d'une telle référence peut se faire de la manière suivante:

```
foreach $k (sort keys %$ref) {
  # $k vaut 'deux', 'trois' puis 'un'
}

foreach $v (sort values %$ref) {
  # $v vaut 1, 2 et 3
}
```

On peut modifier un élément pointé comme ceci:

```
$ref->{'un'} = 'UN';
```

On peut étendre une table de hash référencée.

```
$ref->{'quatre'} = 100;
```

On peut effacer un élément référencé:

```
delete $ref->{'un'};
```

Référence sur une fonction

Cette particularité est appréciable car elle permet entre autres de faire de manière très simple de la génération de code dynamique.

Une référence sur une fonction s'écrit:

```
$func = sub {
             # code de la fonction
           };
```

Pour utiliser la fonction il faut la préfixer du caractère qui désigne (historiquement) la fonction : `&`.

```
&$func();      # lance la fonction
```

Exemple de génération de code dinamique : calculette.pl

```
#!/usr/bin/perl
# Attention ce programme ne gère aucun type d'erreur
# on l'appelle par exemple avec la commande : calculette.pl 2 + 3

# Definition des fonctions
$moins = sub { $_[0] - $_[1] };
$plus = sub { $_[0] + $_[1] };
$multiplie = sub { $_[0] * $_[1] };
$divise = sub { $_[0] / $_[1] };
$calcule = sub { die 'Le code opérateur est inconnu';};

# Attribution de la fonction suivant l'opérateur
if ($ARGV[1] eq '+') { $calcule=$plus; }
elsif ($ARGV[1] eq '-') { $calcule=$moins; }
elsif ($ARGV[1] eq '*') { $calcule=$multiplie; }
elsif ($ARGV[1] eq '/') { $calcule=$divise; }

# Exécution de la fonction et affichage du résultat
print $ARGV[0].' '.$ARGV[1].' '.$ARGV[2].' =
',&$calcule($ARGV[0],$ARGV[2]),"\n";
```

Exemple d'appels de calculette.pl :

```
$ calculette.pl 2 + 3
2 + 3 = 5
$ calculette.pl 2 - 3
2 - 3 = -1
$ calculette.pl 2 test 3
Le code opérateur est inconnu at ./calculette.pl line 10.
```

Annexes

Liens sur Perl

Site	Description
http://www.perl.com/	Site Perl Principal
http://www.perl.org/	Services pour la promotion de Perl
http://www.pm.org/	Site des "Perl Mongers", le groupe d'utilisateurs de Perl
http://www.mongueurs.net/	Site des Mongueurs Perl francophones
http://www.cpan.org/	Compréhensive Perl Archive Network, CPAN
http://search.cpan.org/	Moteur de recherche de CPAN
http://news.perl.org/	Site de nouvelles sur Perl
http://lists.perl.org/	Collection énorme de listes de courrier concernant Perl
http://use.perl.org/	Site de nouvelles et de discussions de la communeauté Perl
http://reference.perl.com/	Belle collection de ressources Perl
http://bugs.perl.org/	Base de données des bougues de Perl
http://www.tpj.com/	*The Perl Journal*, revue technique sur Perl
http://www.perlmonth.com	Magazine mensuel *PerlMonth*