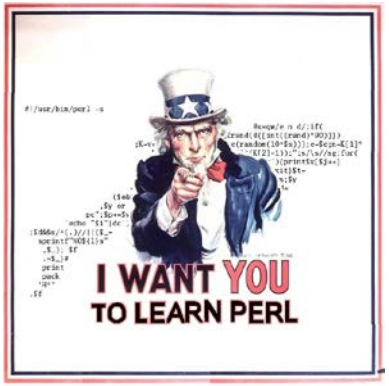
Administration de systèmes et de réseaux

Ecriture de scripts en langage Perl



Qu'est ce que Perl ?

- P.E.R.L. signifie Practical Extraction and Report Language, que l'on pourrait traduire par « 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 interprété
- Nécessité de disposer de l'interprète perl sur la machine pour exécuter des programmes perl
- Logiciel gratuit

Pourquoi Perl est devenu populaire

- Portabilité : Perl existe sur la plupart des plateformes aujourd'hui (Unix, Windows, Mac, VMS, Amiga, Atari ...)
- Disponible gratuitement sur Internet
- Existence d'un nombre impressionnant de librairies et d'utilitaires
- Simplicité : Quelques commandes permettent de faire ce qu'un programme de 500 lignes en C faisait.
- Robustesse : Pas d'allocation mémoire à gérer, chaînes, piles, noms de variables illimités...

Téléchargement de Perl

- Windows :
  https://www.perl.org/get.html

  DWIM Perl for Windows, comprend l'EDI Padre

- Déjà installé sur Linux et MacOS
- Fichiers sources éditables avec un éditeur de code source comme SciTE

Quelle utilisation de Perl ?

A l'origine Perl a été créé pour :

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

=> Perl était essentiellement destiné au monde UNIX

Pourquoi utilise t'on Perl aujourd'hui ?

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

=> Perl n'est plus lié au monde UNIX

Exemple de programme simple

Affichage du nombre de lignes d'un fichier :

```
#!/usr/bin/perl
open (F, 'monfichier');
$i=0;
while (<F>) {
      $i++;
}
close F;
print "Nombre de lignes : $i";
```

Commentaire obligatoire (UNIX): indique l'interpréteur Perl

ouverture d'un fichier en lecture

initialisation du compteur

pour chaque ligne lue…

Incrémentation du compteur

Fermeture du fichier

affichage du contenu du compteur

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.

Types de données

- Perl est un langage faiblement typé : les nombres, les chaînes de caractères, etc., seront tous des scalaires et ne seront différenciés que par leur valeur et par le contexte de leur utilisation
- Il existe principalement trois structures de données : les scalaires, les tableaux et les tables de hachage.
- Un scalaire est une information atomique : nombres, chaines, pointeurs. Les noms des scalaires commencent par un $.
- Un tableau permet de stocker plusieurs scalaires en les indiçant. Les noms de tableaux commencent par @. @t désigne la totalité du tableau t, $t[i] désigne l'élément (scalaire) d'indice i du tableau t. Les indices commencent à 0.

Types de données

- Une table de hachage permet d'associer une chaine (clé unique) à un scalaire. Les noms des tables de hachage commencent par %. %h désigne la totalité de la table de hachage h.
- L'accès à un élément se fait avec un dollar : $h{uneclé} est l'élément de clé uneclé de la table de hachage %h, les accolades { } délimitent la clé.
- Fonctionnellement, on pourrait voir une table de hachage comme un tableau dont les indices peuvent être non-numériques.

Les valeurs des éléments des tableaux et des tables de hachages ne sont pas forcément toutes de même type !

Les scalaires

- Les scalaires sont soit des chaines, soit des numériques
- Exemples de scalaires :

  12 "texte" 'texte' -3.14 3e9
- Les chaînes de caractères ont principalement deux délimiteurs possibles : les " et les ' qui n'ont pas le même rôle :
- Dans une chaîne délimitée par des ", le contenu est **interprété :**

  - dans "Bonjour $prenom" la variable $prenom est substituée par son contenu.

  - dans "il ${prefixe}donn$suffixe", c'est bien la variable $prefixe qui sera utilisée, puis la chaîne donn et enfin la variable $suffixe.
- On notera que comme en shell, $nom peut aussi se noter ${nom} (indispensable pour l'exemple ci-dessus)

Les scalaires

- Pour placer un caractère spécial dans une chaine délimitée par " on le précède d'un \ :

  \" \\ \$ \@

- Dans une chaîne délimitée par des ' aucune interprétation du contenu n'a lieu :
  - 'Bonjour\n' est une chaîne comportant les caractères B o n j o u r \ et n, c'est-à-dire 9 caractères.
  - La chaîne 'Bonjour $prenom' ne comporte pas le contenu d'une hypothétique variable $prenom mais le caractère dollar suivi de la chaine prenom.
  - Les seuls caractères spéciaux à précéder d'un \ dans une chaine délimitée par des ' sont : \' et \\

Déclaration et utilisation des variables

- En Perl, il n'est pas obligatoire de déclarer les variables. Par défaut, l'usage d'une variable la crée ; si c'est un scalaire, elle aura la valeur undef (non définie) ; s'il s'agit d'un tableau ou d'une table de hachage, elle sera vide.
- Pour déclarer une variable on utilise le mot clé my

```
my $x;
my $y = 10;
my $z = "hello";
```

- Exemple d'utilisation de variables:

```
$x = $y + 3;
$prenom = "Jules";
$phrase = "Bonjour $prenom";
```

- Si l'on veut tester qu'une variable scalaire vaut ou non undef, il faut utiliser la fonction defined : if (defined($x))...

Opérateurs, fonctions numériques

- Sur les nombres, les opérateurs classiques sont disponibles : + - / * % (modulo) ** (puissance)
- / est une division réelle, pour faire la division entière il faut prendre la partie entière du résultat : int($x / $y)
- Comme en C et java des raccourcis existent : += -= *= /= %=
- Fonctions mathématiques prédéfinies : sin, cos, exp, log, abs, sqrt, etc.
- Conversion en contexte numérique : les chaînes de caractères représentant exactement un nombre sont converties : "30" + "12" vaut 42.

Opérateurs, fonctions et contexte de chaînes

Les chaînes de caractères ont aussi leurs opérateurs.

- Le point (.) permet de concaténer deux chaînes : l'instruction $x="bon"."jour" a pour effet d'affecter la chaîne "bonjour" à $x
- L'opérateur x est la multiplication pour les chaînes de caractères : "bon"x3 vaut "bonbonbon".
- Fonctions utiles :
  - length($x) renvoie la longueur de la chaîne $x. Par exemple length("bonjour\n") vaut 8 et length('bonjour\n') vaut 9.
  - chop($x) supprime le dernier caractère de la chaîne $x (la variable $x est modifiée). Ce caractère est renvoyé par la fonction : $c = chop($x);
- ...

Fonctions sur les chaînes (suite)

- chomp($x) supprime le dernier caractère de $x s'il s'agit d'une fin de ligne (la variable $x est modifiée), cette fonction peut prendre plusieurs arguments, chacun subira un sort similaire.
- reverse($x) en contexte scalaire, renvoie la chaîne composée des caractères de $x dans l'ordre inverse. Par exemple $v = reverse("bonjour\n") affecte "\nruojnob" à $v.
- substr($x,offset,length) vaut la sous-chaîne de position offset et de longueur length. Les positions commencent à 0. La longueur peut être omise, dans ce cas toute la partie droite de la chaîne est sélectionnée.
- On peut également affecter une valeur à cette fonction :

```
my $v = "salut toi";
substr($v,5,1) = "ation à ";
```

$v vaut alors "salutation à toi".

Fonctions sur les chaînes (suite)

Recherche d'une sous-chaine dans une chaine :

- **`index($chaîne,$sousChaîne,$position)`** renvoie la position de la première occurrence de $sousChaîne dans $chaîne. Le troisième paramètre, s'il est fourni, indique la position du début de la recherche ; sinon la recherche part du début de la chaîne (position 0).

- **rindex($chaîne,$sousChaîne,$position)** effectue la même recherche que la fonction index mais en partant de la fin de la chaîne (la recherche est effectuée de droite à gauche).

Opérateurs de test

- Les booléens n'existent pas en Perl, on utilise les scalaires pour effectuer les tests.
- Les valeurs scalaires fausses sont :
  - 0, c'est-à-dire l'entier valant zéro
  - "0" ou '0', c'est-à-dire la chaîne de caractères ne comportant que le caractère zéro (pas le caractère \0 de code ASCII zéro, mais 0 de code 48)
  - la chaîne vide :"" ou '' (c'est la même chose)
  - undef.
- Toutes les autres valeurs sont vraies, par exemple : 1, -4.2, "blabla", etc. La plus originale est "00" qui vaut l'entier 0 dans les opérations numériques, mais qui est vraie…

Opérateurs de test

- Il existe deux catégories d'opérateurs de test : ceux pour lesquels on impose un contexte numérique aux opérandes et ceux pour lesquels on impose un contexte de chaîne de caractères. Par exemple == teste l'égalité de deux nombres et eq teste l'égalité de deux chaînes ("02"=="2") est vrai alors que ("02" eq "2") est faux
- < teste l'ordre entre nombres, lt teste l'ordre ASCII entre chaînes ; donc (9<12) est vrai alors que (9 lt 12) est faux car 9 est après 1 dans la table ASCII.
- L'option -w de la commande perl permet souvent de repérer ces situations.
  (messages d'avertissement (wanings))

Opérateurs de test

- Opérateurs booléens classiques présents :

  && || ! and or not

	numérique	chaînes
égalité	==	eq
différence	!=	ne
infériorité	<	lt
supériorité	>	gt
inf ou égal	<=	le
sup ou égal	>=	ge
comparaison	<=>	cmp

- L'expression ($x<=>$y) est :
  - positive si $x est un nombre plus grand que $y,
  - négative si $x est un nombre plus petit que $y,
  - nulle si $x et $y sont des nombres égaux.

Structures de contrôle proches de java et C

```
if (condition) {                    # accolades obligatoires
      liste instructions
}
if(condition) {                 ≡   unless(!condition)
    liste instructions
} else {
   liste instructions
}
if( condition1) {                   # pas de switch
    instructions1
} elsif( condition2 ) {
    instructions2
} else {
    instructions3
}
instruction if(condition); # instruction conditionnelle
```

Irérations


```
for( initialisation; condition; incrément ) {
   instructions;
}

while( condition ) {
   instructions;
}

instruction while( condition );

until( condition ) {
   instructions;
}

do {                          do {
   instructions;                 instructions;
} while( condition );         } until( condition );
```

Itérations : les gadgets

- L'instruction next provoque la fin de l'exécution du bloc, le programme évalue directement l'incrément (dans le cas d'une boucle for) puis le test est effectué.
- L'instruction last provoque la fin de la boucle, ni l'incrément ni le test ne sont effectués.
- L'instruction redo provoque le redémarrage du bloc d'instructions sans que la condition ni l'incrémentation ne soient effectuées.

```
while( 1 ) { # 1 est vrai
        $i++;
        last if( $i > 20 );
        next if( $i % 2 != 0 );
        print "$i\n";
}
```

Notion de liste

- Une liste est une suite de valeurs scalaires séparées par des virgules, délimitées par des parenthèses.
- Exemples :

  (2,5,-3)

  (2,'age',"Bonjour $prenom")

  () liste vide

  (1..10) équivalent à (1,2,3,4,5,6,7,8,9,10)

  (1..10,"age","a".."z")
- La liste (1,2,("nom",12),"aaa",-1) n'est pas une liste à 5 éléments dont le troisième serait une autre liste, c'est en fait la liste (1,2,"nom",12,"aaa",-1)
- Pour constituer une liste de listes, il faudra faire usage de références
- L'opérateur de répétition (x) s'applique aussi aux listes : (2,10) x 3 est une liste à 6 éléments valant (2,10,2,10,2,10).

Manipulation de tableaux

- Un tableau est une variable qui peut avoir une liste pour valeur :

```
my @t = (3,'chaine',"bonjour $prenom");
```

- $t[i] **représente l'élément d'indice i du tableau** @t.
- Il est possible d'accéder au dernier élément d'un tableau en utilisant l'indice -1 : $t[-1] est le dernier élément de @t ; de la même façon, $t[-2] est l'avant-dernier, etc.
- Il est possible de connaître l'indice du dernier élément d'un tableau @t grâce à la variable $#t. On a donc $t[$#t] équivalent à $t[-1].
- L'expression scalar(@t) donne le nombre d'éléments du tableau @t.
- $x=@t affecte à $x le nombre d'éléments de @t.

Manipulation de tableaux

- Pas d'erreur (débordement ou autre) si on tente d'accéder à un élément au-delà du dernier. La valeur de cet élément sera simplement undef et le programme continuera.
- Depuis la version 5.6 de Perl, l'instruction exists permet de tester l'existence d'un élément d'un tableau :
  - `if( exists( $t[100] ) ) { … }`
    faux si $t[100] n'existe pas
  - `if( defined( $t[100] ) ) { … }`
    faux si $t[100] existe et vaut undef, soit si $t[100] n'existe pas
- Si @t comporte 100 éléments, l'affectation `$t[1000] = 8;` agrandit le tableau d'autant : les éléments d'indice compris entre 100 et 999 valent undef et scalar(@t) vaut 1001.

Maniputations de tableaux : affectations

- Il est possible d'affecter un tableau à un autre tableau en une seule instruction : `@t = @s;`

  Cette instruction copie le tableau @s dans le tableau @t. Le tableau @t perd ses anciennes valeurs, prend celles de @s et sa taille devient celle de @s ; @t et @s sont alors identiques.

- Affectations mêlant tableaux et listes :

```
($a,$b) = (1,2);
```

Cette instruction affecte une valeur à chacune des variables de la liste de gauche : $a reçoit 1 et $b reçoit 2.

```
($a,$b) = (1);
```

Affectation à $a de la valeur 1 est effectuée et $b est mis à undef (son ancienne valeur est perdue).

Manipulations de tableaux : affectations

- Affectations mêlant tableaux et listes (suite) :

```
($a,$b) = @t;
```

$a et $b reçoivent les premières valeurs du tableau @t : $a en reçoit le premier élément ou undef si @t est vide ; $b reçoit le deuxième élément ou undef si @t ne contient qu'un élément.

```
@t = (1,2);
```

Cette instruction réinitialise le tableau @t (dont les anciennes valeurs sont toutes perdues, y compris celles d'indice différent de 0 et 1) en lui affectant les valeurs de droite : on obtient donc un tableau à deux éléments.

- Affectations de listes (n-uplets) :

($a,$b,$c,$d) = (‘toto’, 12, 3.23, "hello $nom");

- Échange de 2 valeurs sans utiliser d’intermédiaire :

($a,$b) = ($b,$a);

Maniputations de tableaux : affectations

- Affectations mêlant tableaux et listes (suite) :

```
@t = (1,2,"age");
@t2 = (10,@t,20);
```

Le tableau @t2 ne comporte pas trois éléments dont celui du milieu serait lui-même un tableau, mais les cinq éléments, résultat de l'aplatissement du tableau @t dans la liste de droite lors de l'affectation de @t2. Cette affectation a le même résultat que la suivante :

```
@t2 = (10,1,2,"age",20);
```

- Absorption d'une liste par un tableau :

($a,@t) = @s;

La partie gauche de l'affectation est constituée d'une liste comportant une variable scalaire et un tableau : la variable $a reçoit le premier élément du tableau @s et le tableau @t absorbe tous les autres.

En fait dans cette syntaxe, le premier tableau rencontré dans la liste de gauche reçoit tous les éléments restant de la liste de droite. D'éventuelles autres variables qui le suivraient seraient mises à undef s'il s'agit de scalaires et à vide s'il s'agit de tableaux.

Multidéclaration

- Pour déclarer plusieurs variables avec un seul my, l'instruction `my $a,$b;` est incorrecte !
- Pour pouvoir faire cela il faut écrire :

```
my ($a,$b);
```

Les variables $a et $b sont créées et valent undef. Pour leur affecter des valeurs, il faut là aussi utiliser une liste (ou un tableau) :

```
my ($a,$b) = (1,2);
my ($c,$d) = @t;
```

Les mêmes règles que pour l'affectation de listes s'appliquent ici

Tableau @ARGV

- Un tableau qu'il est utile de connaître est @ARGV. Cette variable spéciale est toujours définie et ne nécessite pas de déclaration. Elle contient les arguments de la ligne de commande du programme.
- Toutes les façons de lancer un programme en Perl sont susceptibles d'utiliser @ARGV :
  - `perl -w -e '... code perl...' arg1 arg2 arg3`
  - `perl -w script.pl arg1 arg2 arg3`
  - `./script.pl arg1 arg2 arg3`

  Ces trois programmes sont lancés avec les trois mêmes arguments. Contrairement au langage C, le nom du programme n'est pas contenu dans @ARGV qui ne comporte que les arguments au sens strict.

  La variable spéciale $0 (comme en shell) contient le nom du programme.

Itérateur listes, tableaux : **foreach**

- Forme générale :

```
foreach variable ( liste ) {
   instructions
}
```

Pour chaque élément de la liste, faire…

Exemples :

```
foreach $v (@t) { print "$v\n"; }
foreach $v (32,@t,"age",@t2) { ... }
foreach (@t) {
   print "$_\n";
}
```

$_ désigne les éléments de @t

Comme pour les autres itérations, l'instruction next passe à la valeur suivante sans exécuter les instructions qui la suivent dans le bloc. L'instruction last met fin à l'itération.

Exemple d'usage de foreach et de @ARGV

```
#!/usr/bin/perl -w
use strict;  #oblige à déclarer les variables à l'aide de my
# affichage de tables de multiplications
# La fonction die met fin au programme en affichant un message
die("Usage: $0 <n> <n>\n") if (!defined($ARGV[1]));
foreach my $i (1..$ARGV[0]) {
    foreach my $j (1..$ARGV[1]) {
            printf( "%4d", $i*$j );  # prinft existe !
    }
    print "\n";
}
```

Exemples d'appels :

```
./mult.pl
Usage: ./mult.pl <n> <n>
./mult.pl 5 3
   1   2   3
   2   4   6
   3   6   9
   4   8  12
   5  10  15
```

Fonction de manipulation de tableaux / listes

- Ajout et suppression en tête (à gauche) :
  - La fonction unshift prend en arguments un tableau et une liste de valeurs scalaires, ces valeurs sont ajoutées au début du tableau :

```
my @t = (1,2,3,4);
unshift(@t,5,6);
```

    @t vaut alors la liste (5,6,1,2,3,4).
  - La fonction shift prend un tableau en argument ; elle supprime son premier élément (les autres sont alors décalés) et renvoie cet élément :

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

  - $v vaut alors 1 et @t la liste (2,3,4).
- Inversion de liste :
  - La fonction reverse prend en argument une liste et renvoie la liste inversée :

```
@s = reverse(@t);
```

    @s vaut alors la liste (4,3,2,1) et @t n'est pas modifiée.

Fonction de manipulation de tableaux / listes

- Ajout et suppression en fin (à droite) :
  - La fonction push prend en argument un tableau et une liste de valeurs scalaires ; ces valeurs sont ajoutées à la fin du tableau :

```
my @t = (1,2,3,4);
push(@t,5,6);
```

    @t vaut alors la liste (1,2,3,4,5,6).
  - La fonction pop prend un tableau en argument ; elle supprime son dernier élément et renvoie cet élément :

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

    $v vaut alors 4 et @t la liste (1,2,3).
- Ces fonctions permettent de manipuler facilement des piles et des files

Fonction de manipulation de tableaux / listes

- L'opérateur qw:
  - L'opérateur qw nous permet de créer facilement une liste de chaînes de caractères.

```
@t = qw(Cela est bien facile à faire non ?);
```

@t vaut alors la liste ('Cela', 'est', 'bien', 'facile', 'à', 'faire', 'non', '?').

La chaîne de caractères est découpée selon les espaces, les tabulations et les retours à la ligne. Les délimiteurs les plus souvent utilisés sont les parenthèses ou les slashs :

```
@t = qw/Ou alors comme cela /;
```

- Attention aux erreurs :

```
@t = qw/ attention 'aux erreurs' bêtes /;
```

A pour effet de donner à @t la valeur suivante :

```
("attention","'aux","erreurs'","bêtes")
```

Fonction de manipulation de tableaux / listes

- Joindre les éléments dans une chaîne avec join :

  La fonction join prend en paramètre un scalaire et une liste ; elle renvoie une chaîne de caractères comportant les éléments de la liste, concaténés et séparés par ce premier paramètre scalaire.

  scalaire = join(séparateur, liste);

  Exemples:

```
$s = join(" ",1,2,3);
```

  La variable $s vaut alors la chaîne "1 2 3".

```
$s = join(',',$x,$y,$y);
```

  Les valeurs des trois variables sont jointes en les alternant avec des virgules. Le résultat est affecté à $s.

```
$s = join(" : ",@t);
```

  La variable $s vaut alors la concaténation des valeurs du tableau @t avec " : " pour séparateur.

Fonction de manipulation de tableaux / listes

Découper une chaîne de caractères en liste avec split

- La fonction split prend en paramètres un séparateur et une chaîne de caractères ; elle renvoie la liste des éléments de la chaîne de caractères délimités par ce séparateur :

  liste = split(/séparateur/, chaine);

  Le séparateur est une expression régulière qui est à placer entre slashs

  Exemples :

```
@t = split(/-/,"4-12-1855");
```

  Le tableau @t comporte alors les éléments 4, 12 et 1855.

```
($x,$y) = split(/==/,$v);
```

  Les deux variables auront pour valeur les deux premières chaînes de caractères qui soient séparées par deux signes d'égalité.

Fonction de manipulation de tableaux / listes

Modification avancée de liste avec splice

La fonction splice sert à extraire et modifier un sous-tableau. Elle prend en paramètres une liste, un indice de début et une longueur :

liste2 = splice(liste1, début, long);

splice extrait (enlève) de liste1 la sous-liste de longueur long commençant à l'indice début et la place dans liste2.

Exemple :

```
@t1 = ('Jacob', 'Michael', 'Joshua', 'Matthew', 'Ethan', 'Andrew');
@t2 = splice(@t1, 1, 3);
```

@t2 prend la valeur ('Michael', 'Joshua', 'Matthew')

et la nouvelle valeur de @t1 est ('Jacob', 'Ethan', 'Andrew').

liste2 = splice(liste1, début, long, liste3);

splice extrait (enlève) de liste1 la sous-liste de longueur long commençant à l'indice début et la place dans liste2, puis remplace cette sous-liste par liste3 :

```
@t1 = ('Jacob', 'Michael', 'Joshua', 'Matthew', 'Ethan', 'Andrew');
@t2 = splice(@t1, 1, 3, ('Daniel', 'William', 'Joseph', 'Paul') );
```

@t2 prend la valeur ('Michael', 'Joshua', 'Matthew')

et la nouvelle valeur de @t1 est :

('Jacob', 'Daniel', 'William', 'Joseph', 'Paul', 'Ethan', 'Andrew').

Fonction de manipulation de tableaux / listes

Trier une liste avec sort

- La fonction sort prend en paramètres un bloc d'instructions optionnel et une liste ; elle renvoie une liste triée conformément au critère de tri constitué par le bloc d'instructions. La liste passée en argument n'est pas modifiée.

```
liste2 = sort( liste1 );
liste2 = sort( { comparaison } liste1 );
```

Pendant le tri, le bloc d'instructions comparaison sera évalué pour comparer deux valeurs de la liste ; ces deux valeurs sont localement affectées aux variables spéciales $a et $b.

S'il existe des variables $a et $b dans le programme elles seront localement masquées par ces variables spéciales.

L'expression de la comparaison doit produire une valeur :

- positive, si $a doit être avant $b dans la liste résultat
- négative, si $b doit être avant $a
- nulle, si $a et $b sont égaux (ou équivalents)

=> intérêt des opérations de comparaison cmp et <=>

Fonction de manipulation de tableaux / listes

Trier une liste avec sort

- Les opérateurs de comparaison cmp et <=> permettent de comparer respectivement les chaînes de caractères selon l'ordre lexical et les nombres selon l'ordre numérique.
- Si la fonction sort est appelée sans bloc d'instructions, la liste est triée selon l'ordre lexical.

@s = sort({$a cmp $b} @t); ou @s = sort(@t);

La liste @s a pour valeur la liste @t triée selon l'ordre lexical.

@s = sort({$a <=> $b} @t); Le critère de tri est ici numérique.

@s = sort({length($b) <=> length($a) or $a cmp $b} @t);

les chaines les plus longues en tête et tri lexical pour les mêmes longueurs

@s = sort({ nomfonction($a,$b) } @t);

On peut écrire sa propre fonction de comparaison (à deux arguments) renvoyant une valeur positive, négative ou nulle selon le résultat de la comparaison

Fonction de manipulation de tableaux / listes

Sélectionner des éléments avec grep

- La fonction grep prend en paramètres un critère de sélection et une liste ; elle renvoie la liste des éléments correspondant au critère. La liste passée en argument n'est pas modifiée.
- Le critère de sélection peut être soit une expression régulière, soit un bloc d'instructions.

```
liste2 = grep { sélection } liste1;
```

- Les éléments renvoyés sont ceux pour lesquels l'évaluation du bloc d'instructions a pour valeur vrai. Durant cette évaluation, chacune des valeurs sera localement affectée à la variable spéciale $_ sur laquelle les tests devront être effectués.

```
@t = grep { $_<0 } $x,$y,$z;
```

Affecte à @t les éléments négatifs de la liste.

```
@s = grep { $_!=8 and $_!=4 } @t;
```

Met dans @s les éléments de @t différents de 4 et de 8.

```
@s = grep { nomfonction($_) } @t;
```

Fonction de manipulation de tableaux / listes

Sélectionner des éléments avec grep

- En contexte scalaire, la fonction grep renvoie le nombre d'éléments qui correspondent au critère :

```
$n = grep { .... } @t;
```

- La syntaxe de grep comportant une expression régulière est la suivante :

```
liste2 = grep( /regexp/, liste1 );
```

Les éléments renvoyés seront ceux qui correspondront à l'expression régulière.

Exemple : `@s = grep( /^aa/, @t );`

affecte à @s les éléments de @t qui commencent par deux lettres a.

Plus d'explications sur les expressions régulières sont données plus loin.

Fonction de manipulation de tableaux / listes

Appliquer un traitement à tous les éléments avec map :

- La fonction map prend en paramètres un bloc d'instructions et une liste ; elle applique le bloc à chacun des éléments de la liste (modification possible de la liste) et renvoie la liste constituée des valeurs successives de l'expression évaluée.

  liste2 = map({ expression } liste1);

- La variable spéciale $_ correspond à la valeur de chaque élément de la liste. La valeur de la dernière expression du bloc sera placée dans la liste résultat.
- Exemples :

```
@s = map( { -$_ } @t );
```

Le tableau @s aura pour valeurs les opposés des valeurs de @t.

Fonction de manipulation de tableaux / listes

Appliquer un traitement à tous les éléments avec map

- Exemples (suite) :

  @p = map({ $_."s" } @t);

  Tous les mots de @t sont placés dans @p après leur avoir concaténé un ‘s’.

  @s = map({ substr($_,0,2) } @t);

  Le tableau @s aura pour valeurs les deux premiers caractères des valeurs de @t.

  @s = map({ nomfonction($_) } @t);

  les valeurs renvoyées par la fonction pour chaque élément de @t seront placées dans @s.

- Exemple de modification de liste :

  map({$_= $_*4 } @t);

  Tous les éléments de @t sont multipliés par 4.

Ecriture de fonctions et d'actions

Forme générale d'une fonction Perl :

```
sub maJolieFonction {
my (params formels) = @_;
my … lexique local
... instructions...
return expression;  # valeur renvoyée par la fonction
}
```

- @_ représente la liste des paramètres effectifs
- params formels peut être un tableau
- sub = subroutine = fonction ou action. Une action est une fonction ne renvoyant pas de valeur
- Appel comme dans les autres langages de programmation : nomfonction(liste params effectifs)

Ecriture de fonctions et d'actions

Exemple1 : Factorielle (le grand classique en récursif ☺)

```
sub Fact {
    my ($n) = @_;
    if( $n <= 1) { return 1; }
    else { return $n * Fact($n-1); }
}
print(Fact(5)."\n"); # affiche 120
```

Exemple2 : fonction renvoyant une liste

```
sub f {
    my ($x,$y) = @_;   # deux arguments attendus
    my $m = $x*$y;
    return ($x+$y,$m); # retourne une liste
}
my @t = f(3,5);        # appel de f
print "@t\n";          # affiche 8 15
```

Ecriture de fonctions et d'actions

Portée des variables

- Les variables locales à une subroutine (déclarées au moyen de my) ne sont visibles que dans cette subroutine.
- Dans une subroutine , il est possible d'accéder aux variables définies à la « racine » du programme (c'est-à-dire en dehors de toute subroutine) : il s'agit de variables globales.
- Si une variable locale a le même nom qu'une variable globale, cette dernière est masquée par la variable locale.

```
my $a = 3;
my $b = 8;
my $c = 12;
sub maJolieAction {
    my $a = 5;
    print "$a\n";    # affiche 5
    print "$b\n";    # affiche 8
    $c = 15;         # modification de la variable globale
    print "$c\n";    # affiche 15
}
```

Exemple de fonction

Le crible d'Eratosthène

```
sub crible {   # renvoie la liste des nombres premiers de 1 à n
        my ($n) = @_;
        # Liste initiale :
        my @nombres = (2..$n);  # nombres de 2 à n
        # Liste des nombres premiers trouvés :
        my @premiers = ();
        # Tant qu'il y a des éléments dans @nombres (un tableau
        # en contexte booléen vaut faux s'il est vide)
        while( @nombres ) {
                # On extrait le premier nombre
                my $prem = shift(@nombres);
                # On indique qu'il est premier
                push(@premiers, $prem);
                # On supprime ses multiples (on garde ceux qui ne sont pas multiples)
                @nombres = grep { ( $_ % $prem )!=0 } @nombres;
        }
        unshift(@premiers,1); #on place 1 en tête de la liste des nombres premiers
        # On renvoie la liste des nombres premiers
        return @premiers;
}
```

Tables de hachage (rappel)

- Une table de hachage permet d'associer une chaine (clé unique) à un scalaire. Les noms des tables de hachage commencent par %. %h désigne la totalité de la table de hachage h.
- L'accès à un élément se fera avec un dollar ($) : $h{toto} est l'élément de clé 'toto' de la table de hachage %h, les accolades { } délimitant la clé.
- Fonctionnellement, on pourrait voir une table de hachage comme un tableau dont les indices peuvent être non-numériques.
- Si la clé comporte d'autres caractères que des lettres, des chiffres et le souligné il faut la délimiter au moyen de simples ou de doubles quotes : $h{"Marie-Pierre"} ou $h{'Marie-Pierre'}

Tables de hachage

- Exemples de manipulation :
- Déclaration :

```
my %h = ("Paul" => "01.23.45.67.89",
         "Virginie" => "06.06.06.06.06",
         "Pierre" => "heu..." );
```

- Instructions utilisant h% :

```
$h{Jacques} = "02.02.02.02.02";
print "Tél : $h{Jacques}\n";
$h{'Jean-Paul'} = "03.03.03.03.03";
if( $h{"Jean-Paul"} ne "heu..." ) {
      ...
}
```

Parcours d'une table de hachage

- Il existe trois fonctions permettant de parcourir une table de hachage :
- La fonction keys prend en paramètre une table de hachage et renvoie une liste comportant toutes les clefs de la table. L'ordre des clefs est quelconque, seule l'exhaustivité des clefs est garantie.

```
foreach (keys(%h)) {
   print "Cle=$_ Valeur=$h{$_}\n";
}
```

Ou :

```
foreach my $k (keys(%h)) {
   print "Cle=$k Valeur=$h{$k}\n";
}
```

Parcours d'une table de hachage

- Il existe trois fonctions permettant de parcourir une table de hachage :
- La fonction values prend en paramètre une table de hachage et renvoie une liste comportant toutes les valeurs de la table. L'ordre des valeurs est quelconque, seule l'exhaustivité des valeurs est garantie.

```
foreach my $v (values(%h)) {
  print "Valeur=$v\n";
}
```

- L'ordre des clefs renvoyées par keys et celui des valeurs par values sera le même à condition de ne pas modifier la table de hachage entre temps.

Parcours d'une table de hachage

- Il existe trois fonctions permettant de parcourir une table de hachage :
- La fonction each renvoie un à un tous les couples (clef,valeur) d'une table de hachage. Elle a un comportement un peu spécial du fait qu'il faut l'appeler autant de fois qu'il y a de couples : c'est une fonction avec état : d'un appel à l'autre elle renvoie le couple suivant.
- Utilisation conseillée :

```
while( my ($k,$v) = each(%h) ) {
   print "Clef=$k Valeur=$v\n";
}
```

Table de hachage : Autovivification

- Si on tente de modifier un élément d'une table de hachage qui n'existe pas, il sera créé. Si la clé hello n'existe pas, l'instruction suivante :

```
$h{hello} = "après";
```

crée la clé hello et lui associe la chaine "après".

De même si la clé bye n'existe pas, l'instruction :

```
$h{bye}++;
```

crée la clé bye et lui associe la valeur 1 (0 + 1) ;

si la clé bye existe, elle incrémente la valeur numérique associée de 1 ou, si la valeur n'est pas numérique, associe à la clé bye la valeur 1.

- Cette propriété d'autovivification est bien pratique dans le cas où l'on ne connaît pas les clefs avant de devoir y accéder.

Table de hachage : Autovivification

- Par exemple on peut facilement compter le nombre d'occurrences des mots dans un texte :
  - Supposons que les mots du texte soient déjà dans un tableau
  - Nous allons utiliser chaque mot comme une clé de la table et nous allons ajouter 1 à la valeur de cette clé

```
my @texte = qw( bonjour vous bonjour );
my %comptage = ();
foreach my $mot ( @texte ) {
    $comptage{$mot}++;
}
while( my ($k,$v) = each(%comptage) ) {
    print "Le mot '$k' est présent $v fois\n";
}
```

On affichera :

Le mot 'vous' est présent 1 fois

Le mot 'bonjour' est présent 2 fois

Table de hachage : existence et suppression d'une clé

- L'opérateur exists renvoie vrai si l'élément de table de hachage qu'on lui donne en paramètre existe, faux sinon :

```
if( exists( $h{hello} ) ) { print "La clef 'hello' existe\n"; }
```

- Pour supprimer une clef dans une table de hachage, il faut utiliser l'opérateur delete.
  L'instruction : `delete( $h{hello} );`
  supprime la clef hello de la table %h si elle existe, si elle n'existe pas, elle ne fait rien.
- Pour tester si la table de hachage %h est vide :

```
if( %h eq 0 ) { print "%h est vide\n"; }
if( %h == () ) { print "%h est vide\n"; }
```

Table de hachage et listes

- On peut facilement passer d'une liste (ou tableau) à une table de hachage et inversement :

```
my @t = ("Paul", "01.23.45.67.89", "Virginie",
         "06.06.06.06.06", "Pierre", "heu ...");
my %h = @t;
```

- La première instruction crée un tableau @t initialisé à une liste à 6 éléments. La seconde crée une table de hachage %h initialisée au moyen du précédent tableau. Les valeurs du tableau sont prises deux à deux : la première de chaque couple sera la clef dans la table de hachage, la seconde la valeur. Si le nombre d'éléments de la liste est impair, la dernière clef créée aura undef pour valeur. Si une clef venait à être présente plusieurs fois dans la liste, c'est la dernière valeur qui sera prise en compte dans la table de hachage. On aurait aussi pu écrire :

```
my %h = (  "Paul", "01.23.45.67.89", "Virginie", "06.06.06.06.06",
           "Pierre", "heu ...");
```

Table de hachage et listes

- La conversion de table de hachage vers liste est aussi possible. L'évaluation d'une table de hachage dans un contexte de liste renvoie une liste des clefs et des valeurs, se suivant respectivement deux à deux, dans un ordre quelconque.
- La fonction reverse, peut être employée pour inverser une table de hachage : %h = reverse(%h);
- Les valeurs deviennent les clefs et inversement.
- Si plusieurs valeurs identiques sont présentes le comportement est imprévisible car, certes, lors de la transformation de liste en table de hachage la dernière valeur compte, mais l'ordre est quelconque...
- La variable spéciale %ENV contient les variables d'environnement du système utilisé. $ENV{PATH} contient le path, $ENV{HOME} vaut le nom du répertoire personnel de l'utilisateur qui exécute le programme, etc.

Tranches de tables

- Une tranche de tableau est un sous-ensemble des éléments du tableau.
  - @t[4,3,10] représente une liste à 3 éléments qui est équivalente à ($t[4],$t[3],$t[10]).
  - @t[4..6] représente une liste à 3 éléments qui est équivalente à ($t[4],$t[5],$t[6]).
- Ces notations peuvent être utilisées en partie gauche ou droite d'une affectation, ou pour l'appel d'une fonction qui renvoie une liste :

  (f(params))[bi..bs] ou (f(params))[i1,i2,…,ik]
- De même @h{'clef1','clef2'} est une liste équivalente à ($h{'clef1'},$h{'clef2'}).

Manipulation des fichiers

- Perl dispose d'opérateurs prenant en paramètre un nom de fichier. Leur valeur de retour est souvent booléenne et quelquefois numérique.
- On y retrouve de nombreuses options de la commande test (shell unix):

  -e teste si son paramètre est un chemin valable dans le système de fichier (répertoire, fichier, etc.).

  -f teste si son paramètre est un fichier normal.

  -d teste si son paramètre est un répertoire.

  -l teste si son paramètre est un lien symbolique.

  -r teste l'accès en lecture sur le paramètre.

  -w teste l'accès en écriture sur le paramètre.

  -x teste l'accès en exécution sur le paramètre.

Manipulation des fichiers

- Perl dispose d'opérateurs prenant en paramètre un nom de fichier. Leur valeur de retour est souvent booléenne et quelquefois numérique.
- On y retrouve de nombreuses options de la commande test (shell unix):
  - -o teste si le fichier appartient à l'utilisateur qui exécute le programme.
  - -z teste si le fichier est vide.
  - -s teste si le fichier est non vide ; en fait cet opérateur renvoie la taille du fichier.
- Pour les autres options, faire la commande perldoc -f –X
- Exemples :

```
if( -e "/usr/tmp/fichier" ) { print "Le fichier existe\n"; }
my $taille = -s $file;
```

Manipulation de fichiers : fonction glob

- La fonction glob prend en argument une expression chaine et renvoie une liste de noms de fichier correspondant à l'évaluation de cette expression selon les jokers du shell.
- Par exemple glob('*.java') renvoie la liste des fichiers du répertoire courant ayant l'extension .java
- Il existe une syntaxe plus concise et au comportement identique à cette fonction : l'expression peut être mise entre chevrons. Les deux lignes suivantes effectuent la même opération :

```
@l = glob('/usr/include/*.h');
@l = </usr/include/*.h>;
```

- Après l'exécution d'une de ces deux lignes, le tableau @l contient la liste des noms absolus des fichiers de suffixe .h du répertoire /usr/include.

Ouverture de fichier

- Pour lire ou écrire dans un fichier, il est nécessaire de l'ouvrir préalablement. La fonction effectuant cette opération en Perl se nomme open **et sa syntaxe est la suivante :**

  open(HANDLE, expression);

- Le paramètre HANDLE est l'identifiant ou descripteur du fichier après ouverture. C'est ce descripteur qui devra être fourni aux fonctions de lecture et d'écriture pour manipuler le fichier. La convention veut qu'on le mette toujours en majuscules.
- Le paramètre expression est une chaîne comportant le nom du fichier à ouvrir précédé de zéro, un ou deux caractères indiquant le mode d'ouverture :

Caractère(s)	Mode d'ouverture
aucun	lecture
<	lecture
>	écriture (écrasement)
>>	écriture (ajout)
+>	lecture et écriture (écrasement)
+<	lecture et écriture (ajout)

Ouverture de fichier

- Cette fonction open renvoie une valeur booléenne vrai ou faux indiquant le bon déroulement ou non de l'opération.
- Il est important de tester les valeurs de retour des fonctions manipulant les fichiers. Voici deux exemples de tests de la valeur de retour d'open :

```
if( ! open(FIC,">>data.txt") ) { exit(1); }
open(FIC2, "</tmp/$a") or die("open: $!");
```

- La variable $! contient le message de la dernière erreur survenue, par exemple "No such file or directory" ou "Permission denied", etc. L'utilisateur est donc informé de la cause de la fin du programme.
- Il n'est pas nécessaire en Perl de déclarer un descripteur de fichier, la fonction open valant déclaration.

Lecture, écriture de fichiers

- Une fois un fichier ouvert, il est possible d'écrire et/ou de lire dedans (selon le mode d'ouverture) et de le fermer.
- La lecture des fichiers texte s'effectue typiquement au moyen de l'opérateur chevrons, cette lecture se faisant ligne par ligne.
- L'instruction $l = <FIC>; lit la prochaine ligne disponible du fichier FIC.
- L'instruction @t = <FIC>; 'absorbe' toutes les lignes du fichier dans une liste qui est placée dans le tableau @t.
- Pour itérer sur les lignes d'un fichier :

```
while( defined( $l = <FIC> ) ) {
    chomp $l;
    print "$. : $l\n";
}
```

$l vaut une à une toutes les lignes du fichier. La fonction chomp supprime le dernier caractère s'il s'agit d'un retour à la ligne. La variable spéciale $. vaut le numéro de la ligne courante du dernier fichier lu.

Lecture, écriture, fermeture de fichiers

- Pour écrire dans un fichier, on utilise les fonctions print et printf, elles prennent en premier argument le descripteur de fichier :

```
print( FIC "toto\n" );
printf( FIC "%03d", $i );
```

- Pour fermer un descripteur de fichier (et donc vider les buffers associés), il faut faire appel à la fonction close : `close( FIC );`
- Il existe plusieurs fichiers ouverts automatiquement par Perl dès le lancement du programme :
  - STDIN : l'entrée standard (souvent le clavier).
  - STDOUT : la sortie standard (souvent le terminal). Par défaut print et printf écrivent sur ce flux.
  - STDERR : la sortie d'erreur standard (souvent le terminal). Par défaut warn et die écrivent sur ce flux.

Exécution de commandes shell avec open

- Il est facile en Perl de lancer une commande shell et de récupérer sa sortie standard ou de fournir son entrée standard.
- Pour lire la sortie standard d'un programme, il suffit d'utiliser la syntaxe suivante : open(HANDLE,"commande|");
- Exemple : **`open(FIC1,"ls|");`**

  Les lignes lues via le descripteur de fichier ainsi créé seront celles que la commande aurait affichées à l'écran si on l'avait lancée depuis un terminal.
- La syntaxe open(HANDLE,"|commande") permet de lancer une commande. Les lignes écrites dans le descripteur de fichier constitueront son entrée standard, par exemple :

```
open(FIC3,"|gzip > $a.gz");
open(FIC4,"|mail robert\@bidochon.org");
```

Écrire une table de hachage sur disque avec les fichiers DBM

- Le format DBM est un format de fichier de hachage (clef/valeur) standardisé. Il existe en dehors de Perl.
- En Perl on peut manipuler directement une table de hachage en la liant avec un tel fichier : les valeurs de la table de hachage et du fichier sont synchronisées. Pour y accéder on utilise les fonctions dbmopen et dbmclose.
- Exemple :

```
my %h;
dbmopen(%h,"data",0644) or die($!);
$h{'prenom'} = 'Larry';
dbmclose(%h) or die($!);
```

Utilisation du fichier :

```
dbmopen(%h,"data",0644) or die($!);
print "$h{'prenom'}\n";
dbmclose(%h) or die($!);
```

Fichiers binaires

- La fonction getc renvoie le prochain caractère disponible : $c = getc(FIC);
- La fonction read lit un nombre déterminé de caractères :

```
$tailleLue = read( FIC, $tampon, $tailleALire );
```

- Les données seront placées dans la variable $tampon ; à la fin du fichier, ou s'il y a un problème, le tampon n'est pas complètement rempli.
- Pour écrire des données non textuelles dans un fichier, les fonctions print et printf sont utilisables.
- Noms des fonctions d'entrée/sortie bas niveau en Perl : sysopen, sysread, syswrite et close.

Expressions régulières

- Perl tire des expressions régulières une partie de sa grande puissance pour l'analyse et le traitement des données textuelles.
- Les expressions régulières (regexp) concernent aussi les utilisateurs de grep, sed, Python, PHP, C, C++ et même Java.
- Il y a deux principaux usages des regexp :
  - La correspondance qui est le fait de tester (vérifier) si une chaîne de caractères comporte un certain motif.

    Sa syntaxe est la suivante : m/**motif**/
  - La substitution qui permet de faire subir des transformations à la valeur d'une variable.

    Sa syntaxe est la suivante : s/**motif**/**chaîne**/

Expressions régulières : Bind

- Pour "lier" une variable à une telle expression, il faut utiliser l'opérateur =~ (dit bind en anglais).
- **`$v =~ m/sentier/`** vérifie si la variable $v comporte le mot sentier. La variable est "liée" à l'expression régulière. Cette expression vaut vrai ou faux :

```
if( $v =~ m/sentier/ ) {
    instructions
}
```

- L'opérateur !~ qui équivaut à la négation de =~

if($w !~ m/pieds/) { ... } ≡ if(! ($w =~ m/pieds/)) { ... }

- De la même façon **`$v =~ s/voiture/pieds/;`** remplace la première occurrence de voiture dans la variable $v par pieds, le reste de $v n'est pas modifié.

Caractères dans les expressions régulières

- Dans le cas général, un caractère vaut pour lui-même. Certains caractères ont un rôle particulier dans les, expressions régulières, pour rechercher ces caractères, il faut les dé-spécifier au moyen d'un antislash (\).
- Voici la liste de ces caractères : \ | ( ) [ ] { } ^ $ * + ? .
- Plus le caractère choisi comme séparateur : /
- Par exemple $x =~ m/to\.to/ est vrai si la variable $x comporte les caractères t o . t o contigus.
- Les caractères spéciaux habituels peuvent être utilisés : \n \r \t \f \e
- Le caractère . (point) correspond à un caractère quel qu'il soit (sauf \n)

Caractères dans les expressions régulières

- Le motif [caractères] matche un caractère parmi ceux présents entre crochets.
- Il est possible de définir des intervalles de caractères dans ces ensembles : [a-zA-Z]
- Si le caractère tiret (-) doit être présent dans l'ensemble, il faut le mettre en première ou en dernière position afin de lever toute ambiguïté possible avec un intervalle : [a-z_-]
- Le caractère ^ s'il est placé en début d'intervalle, prend le complémentaire de l'ensemble, il faut le lire "tout caractère sauf..." : [^0-9] [^aeiouy]

Quantificateurs dans les expressions régulières

- Les quantificateurs s'appliquent au motif atomique le précédant dans l'expression régulière. Ils permettent de spécifier un nombre de fois où ce motif peut/doit être présent.
- Par exemple l'étoile * indique que le motif peut être présent zéro fois ou plus : m/a*/ se met en correspondance avec le mot vide, a, aa, aaa, aaaa...

	le motif présent	exemple	mots matchés
*	0 fois ou plus	m/a*/	mot vide, a, aa, aaa…
+	1 fois ou plus	m/a+/	a, aa, aaa…
?	0 ou 1 fois	m/a?/	mot vide ou a
{n}	n fois exactement	m/a{4}/	aaaa
{n,}	au moins n fois	m/a{2,}/	aa, aaa, aaaa…
{,n}	au plus n fois	m/a{,3}/	mot vide, a, aa, ou aaa
{n,m}	entre n et m fois	m/a{2,5}/	aa, aaa, aaaa, ou aaaaa

On remarquera que * est un raccourci pour {0,}
ainsi que + pour {1,}, de même que ? pour {0,1}.

Ensembles dans les expressions régulières

- Un certain nombre de raccourcis représentent des ensembles courants.
  - \d : un chiffre, équivalent à **[0-9]**
  - \D : un non numérique, équivalent à **[^0-9]**
  - \w : un alphanumérique, équivalent à **[0-9a-zA-Z_]**
  - \W : un non alphanumérique : **[^0-9a-zA-Z_]**
  - \s : un espacement, équivalent à **[\n\t\r\f]**
  - \S : un non-espacement, équivalent à **[^ \n\t\r\f]**
- Par exemple, l'expression régulière suivante : **m/[+-]?\d+\.\d+/** permet de reconnaître un nombre décimal, signé ou non : un caractère + ou - optionnel, au moins un chiffre, un point et enfin au moins un chiffre.

Regroupements, Alternatives dans les regexp

Regroupement de motifs

- Pour appliquer un quantificateur conjointement à plusieurs motifs, on utilise les parenthèses.
- L'expression `m/[+-]?\d+(\.\d+)?/` reconnaît des nombres décimaux avec la partie décimale (le point et les chiffres qui le suivent) facultative.

Alternatives

- Il est possible d'avoir le choix entre des alternatives ; il faut pour cela utiliser le signe pipe (|) : l'expression `m/Fred|Paul|Julie/` reconnaît les mots comportant soit Fred, soit Paul, soit Julie.
- L'expression régulière m/(Fred|Paul|Julie) Martin/ reconnaît les trois frères et sœur de la famille Martin.

Assertions dans les expressions régulières

- Une assertion marque une position dans l'expression, elle ne correspond à aucun caractère
- Par exemple, l'accent circonflexe (^) correspond au début de la chaîne. L'expression $v =~ m/^a/ est vraie si la variable $v commence par la lettre a.
- Le dollar ($) correspond à la fin de la chaîne. L'expression $v =~ m/c$/ est vraie si la variable $v se termine par la lettre c.
- Le regroupement au moyen des parenthèses est dit mémorisant. Cela signifie que l'expression matchée par ce regroupement est gardée en mémoire et peut servir à nouveau dans la suite de l'expression.
- Les notations \1, \2 etc. font référence aux sous-chaînes matchées par (respectivement) la première, la deuxième, etc. expression entre parenthèses (maxi : \9)

Regroupements (suite) : motifs mémorisés

- Pour savoir si une variable contient le même mot répété deux fois un utilise l'expression suivante : m/(\w+).*\1/
- \w+ matchera un mot, les parenthèses mémorisent la valeur trouvée, .* permet qu'il y ait un nombre indéfini de caractères quelconques entre les deux occurrences, enfin \1 fait référence à la valeur trouvée du mot par le \w+ précédent.
- Ces motifs mémorisés sont aussi accessibles depuis le second membre d'une substitution au moyen des notations $1, $2, etc.
- Par exemple, l'instruction suivante $v =~ s/([0-9]+)/"$1"/ place des guillemets de part et d'autre du premier nombre de la variable $v : `'sdq 32sq'` deviendra `'sdq "32"sq'`.

Variables définies

- Ces variables spéciales $1, $2 etc. sont aussi accessibles après l'expression régulière.
- Il existe aussi trois autres variables :
  - $& vaut toute la sous-chaîne matchant,
  - $` vaut toute la sous-chaîne qui précède la sous-chaîne matchant,
  - $' vaut toute la sous-chaîne qui suit la sous-chaîne matchant.
- Leur usage est déconseillé car leur usage active des mécanismes particuliers qui ont pour effet secondaire de ralentir fortement la vitesse d'exécution.

Exemple d'usage des variables

```
my $v = "za aa et tfe";
if( $v =~ m/(a+) et ([a-z])/ ) {
   print "$1\n"; # 'aa'
   print "$2\n"; # 't'
   print "$&\n"; # 'aa et t'
   print "$`\n"; # 'za '
   print "$'\n"; # 'fe'
}
```

- L'opérateur de correspondance m// retourne vrai ou faux en contexte scalaire.
- En contexte de liste, cet opérateur retourne la liste des éléments matchés entre parenthèses :

```
my $v = "za aa et tfe";
($x,$y) = ( $v =~ m/(a+) et ([a-z])/ ) # $x = 'aa' et $y = 't'
```

Options dans les expressions régulières

- Si on utilise le / comme séparateur, la lettre m n'est pas obligatoire pour faire un match :

  $v =~ /velo/ est équivalent à $v =~ m/velo/
- Après le dernier séparateur des opérateurs de correspondance ou de substitution il est possible d'indiquer une ou plusieurs options.
- Les syntaxes sont donc : m/motif/options et s/motif1/motif2/options
- Les options permettent de modifier le comportement du moteur d'expressions régulières.

Options dans les expressions régulières

Voici quelques options utiles :

- L'option i rend le motif insensible à la casse : l'expression régulière m/toto/i recherche le mot toto indifféremment en majuscules ou en minuscules. On aurait pu écrire m/[tT][oO][tT][oO]/.
- L'option g permet d'effectuer toutes les substitutions dans la variable. Par exemple, l'expression

  $v =~ s/ +/ /g; remplace dans $v chaque groupe de plusieurs espaces par un seul.
- Dans une substitution, l'option e évalue le membre de droite comme une expression Perl, et remplace le motif trouvé par la valeur de cette expression.

  Par exemple : $s =~ s/(\d+)/fonction($1)/e; remplace le premier nombre trouvé dans la variable $s par la valeur de retour de la fonction appliquée à ce nombre.

Variables dans les motifs, opérateur tr

- Il est tout à fait possible de mettre une variable dans un motif d'une expression régulière. Cette variable sera substituée par son contenu.

  Par exemple :
  - $s = "velo";
  - if($v =~ m/$s$/) { ... }

  La variable sera substituée et la recherche s'effectuera sur 'velo' en fin de chaîne.
- Opérateur tr
  - tr est un opérateur de translation lettre à lettre
  - syntaxe : tr/chaîne1/chaîne2/
  - Les deux chaînes doivent être de la même longueur car cet opérateur va remplacer la première lettre de la première, etc.
  - Il est possible d'utiliser des intervalles : $s =~ tr/a-z/A-Z/; met par exemple le contenu de la variable en majuscules.

Exemple de manipulation de fichier

```
#!/usr/bin/perl -w
use strict;
open(FILE,"<$filename.txt") or die"open: $!";
my($line,@words,$word,%total);
while( defined( $line = <FILE> ) ) {
   @words = split(/\W+/, $line );
   foreach $word (@words) {
         $word =~ tr/A-Z/a-z/;
         $total{$word}++;
   }
}
close(FILE);
foreach $word (sort keys %total) {
   print "$word a été rencontré $total{$word}
   fois.\n";
}
```

Les Références (pointeurs)

- L'opérateur qui permet de prendre la référence d'une variable est l'antislash (\) : \$v est la référence de la variable $v :

```
my $refv = \$v;
```

- La notation $$refv est équivalente à $v tant que $refv pointe vers $v.
- Permet de passer des paramètres par référence et de les modifier dans des subroutines

Les Références (pointeurs)

- L'opérateur qui permet de prendre la référence d'une variable est l'antislash (\) : \$v est la référence de la variable $v :

```
my $refv = \$v;
```

- La notation $$refv est équivalente à $v tant que $refv pointe vers $v.
- Permet de passer des paramètres par référence et de les modifier dans des subroutines

```
sub f {
    my ($ref) = @_;
    $$ref = 0;
}
```

Appel de f : f($refv); ou f(\$v);

Les Références (pointeurs)

- En perl une fonction peut renvoyer la référence d'une variable locale :

```
sub f2 {
    my $w = 43;
    return \$w;
}
```

  - Cette fonction f2 déclare une variable locale $w et renvoie une référence vers cette variable.
  - Ceci est tout à fait légal et sans risque en Perl.

  Appel : my $reff = f2();

  - La variable scalaire $reff devient donc une référence vers une variable scalaire valant 43 qui est l'ancienne variable $w de la fonction f2, et qui n'a plus de nom.
  - La variable locale aurait dû être détruite, mais tant qu'il existe une référence vers la variable, elle est conservée en mémoire.
  - C'est le **garbage-collector** qui la libérera lorsque plus aucune référence sur la variable n'existera.

Références sur tableau

```
my @t = (23, "ab", -54.4);
my $reft = \@t;
```

- @$reft est équivalent à @t. On peut ainsi utiliser la valeur de @t en utilisant $reft
- Pour accéder au ième élément de @t il est maintenant possible d'écrire $$reft[i].
- La notation $reft->[i] est équivalente à $$reft[i].

  $reft->[1] = "coucou"; affecte à l'élément d'indice 1 du tableau pointé par $reft une nouvelle valeur.

Tableau	Référence
t	$reft
@t	@$reft
$t[i]	$$reft[i]
$t[i]	$reft->[i]

Tableau de références de tableaux

- Une référence étant un scalaire, il va nous être possible de stocker une référence comme valeur dans un tableau :

```
my @t1 = ( 16, -33 );
my @t2 = ( "el", 0.3, 4 );
my @t = ( 6, \@t1, \@t2, "s" );
my $r = \@t;
```

Pour accéder aux éléments des différentes profondeurs, suivre les références :

```
print "$r->[0]\n"; # affiche 6
# $r->[1] est une référence vers tableau
print "$r->[1]->[0]\n"; # affiche 16
print "$r->[1][1]\n"; # affiche -33 seconde flèche facultative
# $r->[2] est une référence vers tableau
print "$r->[2]->[0]\n"; # affiche el
print "$r->[2]->[1]\n"; # affiche 0.3
print "$r->[2]->[2]\n"; # affiche 4
print "$r->[3]\n"; # affiche s
```

Références sur table de hachage

```
my %h = ( 'Paul' => 21, 'Julie' => 19 );
my $refh = \%h;
```

- $refh est une référence à %h. %$refh est équivalent à %h.
- Pour accéder à la clé Paul de %h on peut écrire $$refh{Paul} ou $refh->{Paul}.

  $refh->{Jacques} = 33; affecte 33 à l'élément de clé Jacques.

Tableau	Référence
h	$refh
%h	%$refh
$h{Paul}	$$refh{Paul}
$h{Paul}	$refh->{Paul}

Fonction ref

- La fonction ref permet de connaître le type d'une référence. Elle renvoie :
  - "SCALAR" si son argument est une référence sur scalaire,
  - "ARRAY" si son argument est une référence sur tableau,
  - "HASH" si son argument est une référence sur table de hachage,
  - faux si son argument n'est pas une référence.

```
foreach my $p (@$r) {
  if( ref($p) eq "ARRAY" ) {
      print "( ";
      foreach my $v (@$p) { print "$v "; }
      print ")\n";
  } elsif( ref($p) eq "HASH" ) {
      foreach my $k (keys(%$p)) { print "$k : $p->{$k}\n"; }
  } elsif( !ref($p) ) { print "$p\n"; }
}
```

Référence sur fichiers

- Une ouverture de fichier crée une variable dont il est possible de prendre l'adresse :

```
open(FILE,">toto") or die("$!");
my $reff = \*FILE;
```

- On peut stocker une référence vers fichier dans une table de hachage ou dans un tableau.

```
open(FILE,">toto") or die("$!");
my $reff = \*FILE;
print $reff "ok\n";
sub affiche {
    my ($ref) = @_;
    print $ref "ok\n";
}
affiche( $reff );
affiche( \*FILE ); # équivalent
close( $reff );
```

Références sur fonctions

- Une fonction a elle aussi une adresse mémoire. Cette adresse correspond à l'endroit où le code de la fonction est stocké.
- Il est possible de stocker une telle référence dans un tableau ou dans une table de hachage.

```
sub affcoucou  {
    my ($p) = @_;
    print "Coucou $p\n";
}
my $ref = \&affcoucou;
#appel :
&$ref("Larry");
$ref->("Larry"); # équivalent
```

```
#  f a une ref de fonction comme
#  premier paramètre
sub f {
    my ($f,$p) = @_;
    $f->( $p );
}

f( $ref, "Larry" );
f( \&affcoucou, "Larry" ); # équivalent
```

Les Modules de Perl

- Les modules constituent les bibliothèques de Perl. Perl tire sa puissance de la richesse des très nombreux modules existants.
- Peu d'autres langages ont une telle richesse.
- Un module est un ensemble de fonctions, regroupées dans un fichier, qui touchent toutes à un même domaine, un même ensemble de fonctionnalités, un même protocole...
- Certains modules sont installés par défaut avec la configuration de base de Perl, par exemple : Math::Trig, File::Copy, etc.
- Pour ajouter des modules à son installation : commande CPAN et le lien :

  CPAN modules, distributions, and authors (search.cpan.org).

Utilisation d'un module Perl

- Exemple : Math::Trig, un module de fonctions trigonométriques déjà installé.
- Pour visualiser la documentation de ce module, taper la commande perldoc Math::Trig
- Cette commande fonctionne comme la commande man ; taper q pour quitter.
- Pour utiliser un module dans un programme Perl :

```
use NomDuModule;
```

- La ligne use strict; est le chargement d'un module ayant pour rôle de rendre la syntaxe Perl plus coercitive. Le nom des modules de ce type est en minuscule. Ils sont appelés modules pragmatiques. Ils ont pour objet de modifier ou d'étendre la sémantique de Perl.

Utilisation d'un module Perl

- Exemple de code Perl utilisant le module Math::Trig :

```
use Math::Trig;
$x = tan(0.9);
$y = acos(3.7);
$z = asin(2.4);
$pi_sur_deux = pi/2;
$rad = deg2rad(120);
```

- Une fois chargé, un module n'est pas déchargeable.
- File::Copy permet certaines manipulations de fichiers lourdes à mettre en oeuvre avec de simples appels système.

```
use File::Copy;
copy("file1","file2");
copy("Copy.pm",\*STDOUT);'
move("/dev1/fileA","/dev2/fileB");
```

Utilisation d'un module Perl

- Net::FTP qui nous permet d'accéder très simplement aux fonctionnalités d'un client FTP.
- Voici, par exemple de connexion sur un serveur :

```
use Net::FTP;
my $ftp = Net::FTP->new("ftp.cpan.org", Debug => 0, Passive =>1 )
                or die("$!");
$ftp->login("anonymous",'-anonymous@');
$ftp->cwd("/pub/CPAN/");
$ftp->get("ls-lR.gz");
$ftp->quit();
```

Création d'un premier module

- Pour écrire un module, il faut créer un fichier indépendant du ou des scripts qui l'utilisent. L'extension de ce fichier est impérativement .pm : par exemple Utils.pm
- Ce fichier doit être placé dans un des répertoires listés dans la variable @INC, ce peut être .
- Ce fichier doit contenir une première ligne indiquant le nom du module : package Utils;
- Le nom du package doit être le même que celui du fichier.

```
# --- fichier Utils.pm ---
package Utils;
use strict;
sub bonjour {
    my ($prenom) = @_;
print "Bonjour $prenom\n";
}
1;
```

Il est important de ne pas oublier la dernière ligne, celle qui contient 1;

Création et utilisation d'un module

- Pour pouvoir utiliser ce module dans un script, il est nécessaire d'invoquer l'instruction use suivie du nom du module.
- Exemple de l'utilisation du module précédent :

```
#!/usr/bin/perl -w
# --- fichier script.pl ---
use strict;
use Utils;     # chargement du module
Utils::bonjour( "Paul" );
```

- Il est possible de déclarer des variables propres au module, publiques ou privées.
  - Une variable privée se déclare avec my.
  - Une variable publique avec our.

Création et utilisation d'un module

```
# --- fichier Utils.pm ---
package Utils;
use strict;
# variable publique
our $x = 'toto';
# variable privée
my $y = 'toto';
# fonction
sub bonjour {
    # Variable locale
    my ($prenom) = @_;
    print "Bonjour $prenom\n";
}
1;
```

```
#!/usr/bin/perl -w
# --- fichier script.pl ---
use strict;
use Utils;
Utils::bonjour( "Paul" );
print "$Utils::x\n";
```

Le nom complet de la variable publique est donc Utils::x qu'il faut faire précéder d'un $ ce qui donne : **$Utils::x** au final

La valeur **1** est la valeur du chargement du module (valeur de l'instruction use Utils;) ; elle indique si ce chargement s'est bien passé ou non. Une valeur vraie (comme ici 1) indique que le chargement s'est bien déroulé. Il est donc possible de mettre une autre valeur qu'une valeur constante, par exemple un test en dernière instruction pour vérifier si les conditions sont réunies pour l'usage du module.

Création et d'un module aux noms composés

- Les noms composés permettent de regrouper les modules par type d'usages, par exemple Net correspond à tout ce qui concerne le réseau

  (cf. Net::FTP)
- Par exemple, pour un module nommé Truc::Utils, Truc correspond à un répertoire qui doit être présent dans un des répertoires de la variable @INC et le fichier Utils.pm doit être présent dans ce répertoire Truc.
- Bloc BEGIN et END :

  Dans un module, il est possible de prévoir deux blocs d'instructions qui seront exécutés soit dès le chargement du module (bloc **BEGIN) soit lors de la fin de** l'usage du module (bloc **END).**

Fonctions privées d'un module

- Ce n'est a priori pas possible (rien de prévu dans le langage)
- Existence d'une convention : toute fonction ou variable dont le nom commence par un souligné ('_') est privée et ne doit pas être utilisée à l'extérieur du module.
- Les modules CPAN sont bâtis sur ce modèle (c'est beau la confiance…)
- Il existe cependant un moyen d'écrire des fonctions vraiment internes aux modules : il faut déclarer une variable avec my et en faire une référence anonyme vers la fonction :

```
package Utils;
use strict;
my $affiche = sub {
    my ($n,$m) = @_;
    print "$n, $m\n";
};
```

La variable privée $affiche pointe sur une fonction anonyme. Son usage est donc réservé aux fonctions déclarées dans le module :

```
sub truc {
    $affiche->(4,5);
}
```

Documentation des modules

- En Perl la documentation des modules se fait dans le code même du module. Une syntaxe particulière, nommée POD, permet cela. Les instructions POD commencent toujours par le signe égal (=) :

```
=head1 NAME
Utils.pm - Useful functions
=head1 SYNOPSIS
   use Utils;
   bonjour("Paul");
=head1 DESCRIPTION
   Blabla blabla
=head2 Exports
=over
=item :T1 Blabla
=item :T2 Blabla
=back
=cut
```

Les tags **=head1** définissent des en-têtes de premier niveau (des gros titres) et les tags **=head2** définissent des entêtes de deuxième niveau (des sous-titres). Il est de coutume de mettre les premiers exclusivement en majuscules.
Les tags **=over**, **=item** et **=back** permettent de mettre en place une liste.
Le reste du texte est libre. Le tag =cut indique la fin du POD.

Documentation des modules

- Les blocs de POD et les portions de code peuvent alterner : cela est même recommandé de documenter une fonction et d'en faire suivre le code.
- L'apparition en début de ligne d'un tag POD indique la fin temporaire du code Perl et le début d'un bloc de documentation. La fin de ce POD est signalée à l'aide du tag **=cut** et le code Perl reprend.

```
package Utils;
=head1 FUNCTION hello
   This function prints hello.
=cut
sub hello {
   my ($firstName) = @_;
   print "Hello $firstName\n";
}
=head1 FUNCTION bonjour
   This function prints hello in french.
=cut
sub bonjour {
   my ($prenom) = @_;
   print "Bonjour $prenom\n";
}
```

Documentation des modules

- Les blocs de POD et les portions de code peuvent alterner : cela est même recommandé de documenter une fonction et d'en faire suivre le code.
- L'apparition en début de ligne d'un tag POD indique la fin temporaire du code Perl et le début d'un bloc de documentation. La fin de ce POD est signalée à l'aide du tag **=cut** et le code Perl reprend.

```
package Utils;
=head1 FUNCTION hello
   This function prints hello.
=cut
sub hello {
   my ($firstName) = @_;
   print "Hello $firstName\n";
}
=head1 FUNCTION bonjour
   This function prints hello in french.
=cut
sub bonjour {
   my ($prenom) = @_;
   print "Bonjour $prenom\n";
}
```

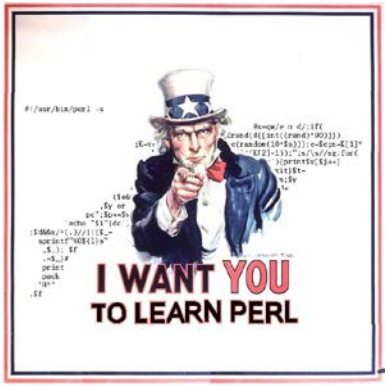
Documentation des modules

- Comment visualiser une telle documentation ?
- Tapez donc perldoc Utils et sa documentation apparaît au format man comme tout bon module CPAN :

```
NAME
        Utils.pm - Useful functions
SYNOPSIS
        use Utils;
        bonjour("Paul");
DESCRIPTION
        Blabla blabla
        Exports
        :T1 Blabla
        :T2 Blabla
FUNCTION hello
        This function prints hello.
FUNCTION bonjour
        This function prints hello in
french.
```

Programmation par Objets en Perl

Extension des modules pour faire de l'Objet



Programmation objet

- En Perl, une classe est un module et un objet est une référence associée à cette classe.
- Dans le constructeur on crée une référence vers une table de hachage et on l'associe au package en question ; lors de cette association, on dit que l'on bénit (bless en anglais) la référence.
- Les champs de l'objet seront en fait stockés dans cette table de hachage, sous forme de la clé pour le nom du champ et de la valeur pour la valeur du champ.

Programmation objet : écriture d'un constructeur

- Nous définissons un package Vehicule dans un fichier Vehicule.pm

```
1. # --- fichier Vehicule.pm ---
2. package Vehicule;
3. use strict;
4. sub new {
5.     my ($class,$nbRoues,$couleur) = @_;
6.     my $this = {};
7.     bless($this, $class);
8.     $this->{nb_Roues} = $nbRoues;
9.     $this->{couleur} = $couleur;
10.    return $this;
11. }
12. 1; # À ne pas oublier...
```

Par convention les noms de classes/modules commencent par une majuscule (comme en java). Un constructeur en Perl est une simple fonction renvoyant un objet. Le choix du nom pour cette fonction est totalement libre ; il est courant de l'appeler new mais rien ne nous y oblige.
Cette fonction prend en premier paramètre le nom de la classe ; cela semble superflu, mais on verra qu'il n'en est rien.

Programmation objet : écriture d'un constructeur

- Nous définissons un package Vehicule dans un fichier Vehicule.pm

```
1. # --- fichier Vehicule.pm ---
2. package Vehicule;
3. use strict;
4. sub new {
5.     my ($class,$nbRoues,$couleur) = @_;
6.     my $this = {};
7.     bless($this, $class);
8.     $this->{nb_Roues} = $nbRoues;
9.     $this->{couleur} = $couleur;
10.    return $this;
11. }
12. 1; # À ne pas oublier...
```

Ligne 6 : création d'une référence anonyme vers une table de hachage vide {} stockée dans une variable scalaire nommée $this (nom choisi arbitrairement).
Ligne 7: cette référence est liée au package (à la classe) $class. Cette variable $class vaudra ici Vehicule. L'opérateur bless associe le package à la référence.
Lignes 8 et 9, les champs nb_Roues et couleur sont initialisés. Le champ d'un objet n'est rien d'autre qu'une entrée dans la table de hachage constituant l'objet.

Programmation objet : écriture d'un constructeur

- Remarque : le nombre, le nom et le contenu des champs peuvent varier d'une instance de la classe à une autre instance de cette même classe.
- Libre au programmeur de faire ce qu'il veut :
  - Pour vraiment programmer objet de façon formelle, il faut respecter les habitudes de ce type de programmation qui veut que toutes les instances d'une même classe aient les mêmes champs ;
  - *Si on ne tient pas à respecter ces contraintes, il est possible de faire ce que l'on veut de chacun des objets (déconseillé)*

Appel du constructeur

Exemples d'appel du constructeur

```
use strict;
use Vehicule;
# Nous pouvons maintenant utiliser le constructeur
# que nous avons défini

my $v = Vehicule->new( 2, "bleu" );
my $v2 = Vehicule->new( 4, "rouge" );

# Il est aussi possible de faire usage de la syntaxe suivante
my $v3 = new Vehicule( 2, “vert" );
```

Plusieurs constructeurs

- Il suffit de créer une autre fonction qui renvoie une référence « bénie » (autre nom)

```
sub nouveau {
    my ($class,$couleur) = @_;
    my $this = {};
    bless($this, $class);
    $this->{nb_Roues} = 0;
    $this->{couleur} = $couleur;
    return $this;
}
```

- Utilisation du constructeur

```
my $v2 = Vehicule->nouveau("bleu");
# ou
my $v3 = nouveau Vehicule("bleu");
```

Ecriture d'une méthode

```
sub roule {
 my ($this,$vitesse) = @_;
 print("Avec $this->{nb_Roues} roues, je roule à    $vitesse.\n");
}
```

Cette fonction déclarée dans le fichier **Vehicule.pm** a donc pour premier paramètre l'objet sur lequel elle est appelée.

Note: rien n'oblige le programmeur à nommer cette variable $this ; la seule contrainte est sa première place parmi les paramètres.

Utilisation de la méthode dans un script :

```
$v->roule( 15 );
```

La fonction appelée sera celle qui a pour nom roule définie dans le package lié à la référence $v lors de sa bénédiction.

Remarque importante

- Les champs d'un objet étant de simples clef/valeur d'une table de hachage dont on dispose d'une référence dans le script, ils y sont accessibles.
- Dans le script, nous pouvons écrire:

```
foreach my $k (keys %$v) {
    print "$k : $v->{$k}\n";
}
```

- Dans le script, on peut accéder sans restriction à l'ensemble des champs de l'objet.
- Faute de champs privés ou de méthodes privées on utilise la convention consistant à faire débuter un nom privé par le caractère '_'

Composition

```
# --- fichier Garage.pm ---
package Garage;
use strict;
sub new {
    my ($class,$places) = @_;
    my $this = {};
    bless($this, $class);
    $this->{places} = $places;
    $this->{vehicules} = [];
    return $this;
}
1;
```

- Ce constructeur prendra en paramètre le nombre de places disponibles du garage ; cette valeur est enregistrée dans le champ places. Le champ vehicules comportera la liste des véhicules ; pour cela elle est initialisée à la référence anonyme vers une liste vide.

Composition

La méthode ajoute se charge d'ajouter les véhicules dans la limite du nombre de places disponibles :

```
# --- fichier Garage.pm ---
sub ajoute {
   my ($this,$vehicule) = @_;
   if ( @{$this->{vehicules}} < $this->{places} ) {
      push( @{$this->{vehicules}}, $vehicule; )
      return 1;
   } else {
      return 0;
   }
}
```

Cette méthode prend en paramètre une référence vers un véhicule. Elle compare la longueur
de la liste des véhicules au nombre total de places

Exemple de script utilisant des classes

```
# --- fichier contenant le script ---
use Garage;
use Vehicule;
   my $g = Garage->new(3);
   my $v = new Vehicule( 2, "bleu" );
   $g->ajoute( $v ) or die("ajoute: plus de place");
   $g->ajoute( Vehicule->new( 4, "vert" ) )
                     or die("ajoute: plus de place");
   $g->ajoute( Vehicule->new( 1, "jaune" ) )
                     or die("ajoute: plus de place");
```

Destruction d'un objet

- Un objet est détruit dès qu'aucune référence ne pointe vers cet objet. La ligne suivante, par exemple, libère la place mémoire occupée par l'objet Vehicule référencé par $v2 :

```
$v2 = undef;
```

- À cet instant, Perl se rend compte que l'objet en question n'est plus accessible, la mémoire sera donc automatiquement libérée par le mécanisme du garbage collector.
- Il existe une méthode spéciale, dont le nom est réservé, qui est appelée lors de la destruction d'une instance d'un objet. Il s'agit de la méthode DESTROY. Cette méthode sera appelée (si elle existe dans la classe) par le garbage collector juste avant la libération de la mémoire de l'objet.

À vous de jouer !