Guide Perl

— débuter et progresser en Perl —

Sylvain Lhullier
http://formation-perl.fr/

Version 1.3.20
-
septembre 2016

Table des matières

Licence

Version 1.3.20 - septembre 2016

Vous trouverez la dernière version de ce document à cette adresse :
`http://formation-perl.fr/guide-perl.html`
Ce document est y disponible en version HTML, PDF et ePUB.

Introduction

Ce guide Perl sert de support à la formation Perl. C'est une introduction au langage initialement écrite pour *Linux Magazine France* et parue dans les numéros de juillet 2002 à février 2003 puis réédité au printemps 2004 dans les Dossiers Linux 2. Depuis, ce document est régulièrement mis à jour.

Ce langage très riche et puissant est une boîte à outils fort utile dans de nombreuses situations : administration système, manipulation de textes (mail, logs, linguistique, génétique), programmation réseau (CGI, `mod_perl`, etc.), bases de données, interfaces graphiques, etc. Ses nombreuses bibliothèques le rendent vite irremplaçable aux yeux de ceux qui en acquièrent la maîtrise. La prise en main du langage est facilitée par de nombreux rapprochements possibles avec le C, le shell ou awk. Sa conformité POSIX en fait un allié indispensable à l'administrateur système.

Ce document a la délicate ambition de s'adresser à la fois au programmeur débutant et à celui qui connaîtrait bien le C ou le shell. Que le premier me pardonne de faire des comparaisons avec d'autres langages et de taire peut-être certains points qui me semblent évidents. Que le second m'excuse de passer à son goût trop de temps à expliquer des notions qui lui semblent simples ; les choses se corseront au fur et à mesure de la lecture...

Le début du document aborde des notions importantes en Perl et nécessaires pour bien comprendre la suite. Vous serez sans doute un peu déçu de ne pas faire des choses extrêmement puissantes immédiatement, mais patience : qui veut aller loin ménage sa monture.

Pour vous mettre en appétit, voici un petit exemple de la concision de Perl et de sa puissance :

```
my @r = qw(Un programme Perl est 5 fois plus rapide a ecrire);
map { tr/A-Z/a-z/; s/\d//g; } @r;
foreach (sort grep !/^$/, @r) { print "$_\n"; }
```

Ce programme crée une liste de mots (la phrase de la première ligne), transforme les majuscules de ces mots en minuscules, supprime les chiffres appartenant aux mots, supprime les mots vides et affiche la liste des mots ainsi transformés dans l'ordre lexical. Et dites-vous que vous aurez en main toutes les notions nécessaires avant la fin de la lecture du document...

Perl est un langage de haut niveau, qui a la prétention de combiner les avantages de plusieurs autres langages. Première facilité, il gère lui-même la mémoire (ramasse-miettes, pas de limite de buffers, pas d'allocation à faire, etc.). De plus, les tableaux, les listes et les tables de hachage sont natifs, ils sont intégrés à la grammaire même du langage. Récursivité, modularité, programmation objet, accès au système et au réseau, interface avec le C, avec (g)Tk, avec Apache sont aussi au menu. Et souvenez-vous que l'une des devises de Perl est : *there is more than one way to do it* (il y a plus d'une façon de le faire).

Chapitre 1

Premier pas

1.1 Exécuter un programme en Perl

Il existe principalement deux types de langages : les langages compilés et les langages interprétés. Pour les premiers (on retrouve par exemple dans cette catégorie le C et le C++), il existe deux phases distinctes : la compilation des sources par un compilateur (`gcc` par exemple) puis l'exécution du programme ainsi obtenu par le système. Pour les seconds (les shells par exemple), il n'y a pas de phase de compilation, un interpréteur va lire le code et directement agir en fonction de celui-ci.

Perl est un langage à la fois interprété et compilé. Il n'y a pas de phase intermédiaire de compilation, car l'interpréteur (qui se nomme `perl` en minuscules, alors que le langage prend une majuscule) compile le code sans que le programmeur ne s'en rende compte, puis l'exécute. L'interpréteur se charge donc à la fois de la compilation et de l'exécution.

Il existe trois façons distinctes de faire tourner un programme Perl :

- mettre du code en ligne de commande. On peut écrire ceci dans un shell :
  `perl -e 'print("Salut Larry\n");'`
  Le programme `perl` est lancé avec du code Perl comme argument, directement sur la ligne de commande (option `-e`). En anglais, on appelle cela un one-liner (pas de traduction établie en français, peut-être monoligne, uniligne, soliligne...). Le code sera alors exécuté : la fonction `print` affiche son argument.
  `Salut Larry`
  Ce n'est pas la façon la plus courante d'écrire du Perl, mais c'est une manière facile et rapide de faire un petit calcul ou de faire appel à une fonction Perl.
- la seconde manière de faire est de créer un fichier `salut.pl` contenant :
  ```
  print("Salut Larry\n");
  ```
  Puis de lancer la commande suivante depuis un shell :
  `perl salut.pl`
  Le programme `perl` est lancé avec le nom du fichier en argument. Il va alors lire le contenu de ce fichier et l'interpréter comme du code Perl ;
- la troisième façon de faire est de créer un fichier `salut2.pl` contenant :
  ```
  #!/usr/bin/perl
  use strict;
  use warnings;
  print("Salut Larry\n");
  ```

La première ligne est le *shebang*, bien connu des habitués des scripts en shell. Cette ligne (qui doit toujours être la première du fichier) indique au système d'exploitation le chemin de l'exécutable à lancer pour interpréter ce fichier. Le premier caractère doit être un dièse, ce qui a pour effet que cette ligne est considérée comme un commentaire Perl par l'interpréteur. Ensuite un point d'exclamation. Puis le chemin absolu vers l'exécutable `perl` (à adapter selon votre installation, voir ce que répond `type perl` ou `which perl`).
Les deux lignes suivantes sont des *pragma* qui rendent le langage moins permissif.
Le pragma `use strict;` permet de rendre le langage moins permissif, notamment en nous obligeant à déclarer nos variables. Je vous conseille de toujours utiliser ce pragma.
Le pragma `use warnings;` est positionné dans le but que l'interpréteur affiche des messages d'avertissement (*warnings*) à différents propos : il indique les variables utilisées une seule fois ou utilisées avant d'être initialisées, il signale les redéfinitions de fonctions, etc. Je vous conseille donc de toujours utiliser ce pragma.
Il faut maintenant rendre ce fichier exécutable et le lancer :

```
chmod +x salut2.pl
./salut2.pl
```

Grâce à la ligne de shebang, le système le reconnaît donc comme un programme nécessitant un interpréteur pour être exécuté. Cet interpréteur est lancé avec pour paramètres les options fournies dans le shebang ainsi que le nom du fichier.
Cette dernière façon de faire est sans doute la plus courante dès que l'on écrit un programme qui sera utilisé plusieurs fois.

Avant de continuer, quelques commentaires sur la syntaxe Perl. Excepté dans les chaînes de caractères, la présence ou l'absence d'espaces, de sauts de ligne ou de tabulations ne change pas le comportement du programme, l'indentation est libre. Comme on vient de le voir, une instruction Perl est terminée par un point-virgule. Les fonctions peuvent prendre leurs arguments entre parenthèses (comme en C) ; il faut aussi savoir que l'on peut se passer de ces parenthèses. On aurait pu écrire :

```
print "Salut Larry\n";
```

Les deux syntaxes sont tout à fait valides. Dans les cas simples (appel d'une fonction avec un ou deux paramètres), on pourra omettre les parenthèses. Dans le cas de combinaisons plus complexes (plusieurs fonctions appelées en paramètre d'autres fonctions, listes en argument...), je vous conseille de les mettre pour lever toute ambiguïté possible.

Un commentaire commence par un dièse (`#`) et se termine en fin de ligne (comme en shell). On constitue un bloc d'instructions en les regroupant dans des accolades `{}` comme en C. Des exemples suivront.

1.2 Les types de données

Perl est un langage faiblement typé, ce qui signifie qu'une donnée n'aura pas spécialement de type : les nombres, les chaînes de caractères, les booléens, 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. Chaque structure de données est liée à un caractère spécial (lire la suite).

Un scalaire est une donnée atomique. Par exemple, ce pourrait être une chaîne de caractères (suite de caractères) ou un nombre (entier ou flottant). On verra plus tard que les références

(c'est-à-dire les pointeurs de Perl) sont des scalaires, même s'ils sont un peu spéciaux. Les variables scalaires sont précédées d'un dollar (`$`) : `$x` est donc une variable scalaire.

Les tableaux permettent de stocker plusieurs scalaires en les indiçant. De la même façon qu'en C, on pourra demander le ième élément d'un tableau, i étant un entier. L'accès à un élément sera en temps constant, il ne dépendra pas de son indice dans le tableau. Les variables de type tableau sont précédées d'une arobase (`@`) : `@t` est donc une variable de type tableau. Lorsque l'on utilise la syntaxe `@t`, on désigne la totalité du tableau ; si nous voulons parler du ième élément, nous allons manipuler un scalaire, il faudra donc préfixer par un dollar : `$t[4]` est l'élément d'indice 4 dans le tableau `@t`, les crochets `[]` délimitant l'indice (nul besoin de mettre l'arobase, Perl sait grâce aux crochets qu'il s'agit d'un élément de tableau).

Une table de hachage en Perl est une structure de données permettant d'associer une chaîne de caractères à un scalaire ; on parle de clefs et de valeurs : une valeur est associée à une clef. Naturellement, dans une même table de hachage les clefs sont uniques ; les valeurs, par contre, peuvent être tout à fait quelconques. Les variables de type table de hachage sont précédées d'un caractère pourcent (`%`) : `%h` est donc une variable de type table de hachage. De la même façon que pour les tableaux, `%h` représente la totalité de la table de hachage ; accéder à un élément se fera avec un dollar : `$h{uneclef}` est l'élément de clef `uneclef` de la table de hachage `%h`, les accolades `{}` délimitant la clef. Fonctionnellement, on pourrait voir une table de hachage comme un tableau dont les indices peuvent être non numériques.

Nous reviendrons sur ces types de données tout au long du document.

1.3 La notion de contexte

Chaque opération en Perl est évaluée dans un contexte spécifique. La façon dont l'opération se comportera peut dépendre de ce contexte. Il peut jouer un rôle important sur le type des opérandes d'une expression et/ou sur le type de sa valeur. Il existe deux contextes principaux : le contexte scalaire et le contexte de liste.

Par exemple, une affectation d'une expression à une variable de type scalaire évaluera cette expression dans un contexte scalaire ; de la même façon, une affectation à une variable de type liste évaluera le membre droit en contexte de liste. Autre exemple que je détaillerai un peu plus loin, les opérateurs de test imposent un contexte précis à leurs opérandes.

Certains opérateurs et fonctions savent dans quel contexte ils sont appelés et renvoient un scalaire ou une liste selon ce contexte d'appel. La fonction `grep` en est un bon exemple (nous verrons cela lorsque nous parlerons des listes). On peut forcer le contexte d'une expression au contexte scalaire en utilisant l'opérateur `scalar()`.

Il existe plusieurs contextes scalaires : le contexte de chaînes de caractères, le contexte numérique et le contexte tolérant. Par exemple une addition impose un contexte numérique à ses deux opérandes ; cela signifie que les opérandes sont transformés en nombres, quels que soient leur type et leur valeur (reste ensuite au programmeur à savoir ce que vaut une liste ou une chaîne quelconque en contexte scalaire...). Le contexte de chaînes est un contexte où, comme son nom l'indique, les scalaires seront considérés comme des chaînes de caractères. Le contexte tolérant n'est ni de chaînes ni numérique, il est simplement scalaire.

Il existe aussi un contexte vide (*void context* in English) qui correspond au fait que la valeur d'une expression est ignorée. C'est par exemple le contexte utilisé lorsque l'on appelle une fonction sans récupérer sa valeur de retour, comme l'appel à la fonction `print` dans la section précédente. Ce contexte n'apporte pas grand chose au programmeur, mais permet à l'interpréteur

Perl appelé avec `use warnings;` de prévenir en cas d'usage d'une expression sans effet de bord en contexte vide (c'est-à-dire une expression qui ne fait rien et dont on ne se sert pas). Par exemple, la ligne d'instructions suivante :

```
"Bonjour";
```

provoque le message suivant :

```
Useless use of a constant in void context at prog.pl line 5.
```

Vous aurez l'occasion de manipuler les contextes tout au long de cette introduction au langage. Même si cela n'est pas forcément explicite à chaque instant, le contexte est souvent important.

Chapitre 2

Les scalaires

Les scalaires sont le type de données atomique de Perl, dit autrement un scalaire est une donnée atome. Cela signifie que la granularité de données ne va pas au-delà.

Comme dit précédemment, une variable scalaire peut contenir une chaîne de caractères (`String` en Java et autres) ou un nombre (entier ou nombre à virgule flottante : `int` ou `float` en C, C++, etc.) ; je ne rentrerai pas dans l'explication de ce "ou". Voici des exemples de scalaires corrects : `12 "texte" 'texte' -3.14 3e9`

Contrairement au C où le caractère `\0` de code ASCII 0 (zéro) est le marqueur de fin de chaîne, en Perl les chaînes de caractères peuvent sans souci contenir ce caractère : `"a\0f"` est une chaîne comportant trois caractères. On n'aura donc aucun mal à traiter des fichiers binaires en Perl.

2.1 Les délimiteurs de chaînes de caractères

Les chaînes de caractères ont, comme en shell, principalement deux délimiteurs possibles : les doubles-quotes (`"`) et les simples quotes (`'`). Elles n'ont pas le même rôle :

- dans une chaîne délimitée par des doubles-quotes, le contenu est *interprété* :
  `"Bonjour\n"` est une chaîne suivie d'une fin de ligne. De la même manière `"\t"` est une tabulation (il existe d'autres caractères spéciaux).
  Dans `"Bonjour $prenom"` la variable `$prenom` est *substituée* par son contenu ; c'est-à-dire que ce scalaire contiendra la chaîne `Bonjour`, suivie d'une espace, suivie du contenu de la variable `$prenom`.
  S'il faut accoler un texte immédiatement après une variable, on utilisera les accolades pour délimiter le nom de la variable ; par exemple 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 ces accolades n'ont rien à voir avec celles des tables de hachage.
  Certains caractères doivent être "protégés" avec un anti-slash (\) si on veut les faire apparaître tels quels dans la chaîne de caractères ; ce sont les quatre suivants : `" $ @ \`. La chaîne `"\$v"` ne contient donc pas la valeur d'une supposée variable `$v` mais contient le caractère dollar et le caractère `v` ;
- dans une chaîne délimitée par des simples quotes, 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 neuf caractères (notez que `'\n'` comporte deux caractères).

La chaîne `'Bonjour $prenom'` ne comporte pas le contenu d'une hypothétique variable `$prenom` mais le caractère dollar suivi de la chaîne `prenom`.
Puisque les variables ne sont pas substituées, les caractères à protéger sont moins nombreux. Le caractère `'` a besoin d'être précédé d'un anti-slash pour apparaître tel quel dans une chaîne délimitée par de simples quotes. Il en est de même pour le caractère `\` si celui-ci est suivi d'un autre caractère `\`

Les nombres n'ont quant à eux pas besoin de délimiteurs pour être manipulés : `$x = 10.2` affecte le nombre 10,2 à la variable `$x`.

2.2 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` (lire plus loin) ; s'il s'agit d'une liste ou une table de hachage, elle sera vide.

Pour d'évidentes raisons de relecture et pour éviter des erreurs bêtes, je vous conseille de toujours déclarer vos variables avant de les utiliser (sauf peut-être dans le cas de scripts de quelques lignes). Pour déclarer une variable, il nous faut utiliser `my` :

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

Nous venons ici de déclarer trois variables scalaires. Ces variables seront visibles (accessibles) dans toute la suite du bloc ainsi que dans les sous-blocs (comme en C) ; comme on s'y attend, elles ne le seront par contre pas dans les fonctions appelées depuis ces blocs. Le placement des déclarations est libre dans le bloc (comme en C++), il n'est pas nécessaire de les mettre en début de bloc.

Voici quelques exemples d'utilisation de variables (on suppose qu'elles sont déjà déclarées) :

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

Cette dernière ligne affichera à l'écran `Bonjour Jules` suivi d'un caractère de nouvelle ligne. Les habitués du shell noteront bien qu'une variable est toujours précédée de son dollar même si elle est à gauche d'un égal d'affectation.

2.3 La valeur `undef`

C'est une valeur particulière signifiant « non défini ». C'est aussi la valeur par défaut des variables scalaires non initialisées : `my $x;` est équivalent à `my $x=undef;` On peut affecter cette valeur à une variable après son initialisation : `$x=undef;` ou `undef($x);`

Si l'on veut tester qu'une variable scalaire vaut ou non `undef`, il faut utiliser la fonction `defined` : `if(defined($x))...` Ce test est vrai si `$x` est définie, c'est-à-dire si elle ne vaut pas `undef`. Une erreur classique est d'écrire : `*incorrect* if($x!=undef) *incorrect*` Ne surtout pas tenter de comparer une variable à `undef`, car cela ne fait pas ce qu'on attend.

La valeur `undef` est une valeur fausse pour les tests. Le test `if( $x ) ...` est faux si `$x` est non définie. Mais comme on le verra plus tard, il est également faux si `$x` vaut 0 (zéro) ou bien la chaîne vide. Donc un test `if( $x ) ...` est potentiellement dangereux. Pour tester si une variable est définie, une seule bonne façon : `if(defined($x))...`

2.4 Opérateurs, fonctions et contexte numériques

Sur les nombres, les opérateurs classiques sont disponibles : `+ - / * %` ; ce dernier opérateur `%` est le modulo, c'est-à-dire le reste de la division entière du premier opérande par le second. Notez que la division effectuée par l'opérateur `/` n'est pas une division entière, mais une division réelle, cela même si ses opérandes sont entiers (2/3 vaut 0.6666...) ; si vous voulez effectuer une division entière, il vous faut tronquer le résultat de la division précédente avec `int()` : l'expression `int($x/$y)` vaut le quotient de la division entière de `$x` par `$y` (pour des nombres positifs).

Des raccourcis existent : `+= -= *= /= %=`. Ces opérateurs sont à la fois une opération arithmétique et une affectation : `$x+=3` est équivalent à `$x=$x+3` mais en plus synthétique : on ajoute 3 à `$x`. L'instruction `$y*=5` multiplie `$y` par 5.

Il existe aussi des auto-incrémenteurs et des auto-décrémenteurs : `++` et `--` qui peuvent être placés avant ou après une variable : ils ajoutent ou déduisent 1 à cette variable. `$x++` à le même effet que `$x+=1` ou que `$x=$x+1`.

L'opérateur `**` correspond à la puissance : `2**10` vaut 1024.

Les fonctions suivantes manipulent les nombres :

- `sin($x) cos($x)` renvoient le sinus et le cosinus de `$x`.
- `exp($x) log($x)` renvoient e puissance `$x` et le logarithme en base e de `$x`.
- `abs($x)` renvoie la valeur absolue de `$x`.
- `sqrt($x)` renvoie la racine carrée de `$x`.

Voici quelques règles de conversion en contexte numérique. Les chaînes de caractères représentant exactement un nombre sont converties sans problème ; `"30" + "12"` vaut 42. Dans tous les autres cas (énumérés dans ce qui suit), le pragma `use warnings;` provoquera un message d'avertissement. Les valeurs scalaires commençant par un nombre sont converties en ce nombre : `"34.2blabla"` vaudra 34,2. Les autres valeurs scalaires (y compris `undef`) sont converties en 0. Conclusion : utilisez toujours `use warnings;` !

2.5 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` (pas de gestion de la mémoire à effectuer).

Cet opérateur est, entre autres cas, utile lorsque certaines parties de la chaîne sont les valeurs de retour de fonctions ; en effet, il suffit souvent d'utiliser les substitutions effectuées dans les chaînes délimitées par des doubles-quotes pour concaténer deux chaînes.

L'opérateur `x` est la multiplication pour les chaînes de caractères : `"bon"x3` vaut `"bonbonbon"`. Fort sympathique...

Les raccourcis suivants peuvent être utilisés : `.= x=` L'expression `$x.=$y` est équivalente à `$x=$x.$y` et concatène donc `$y` à la fin de `$x`.

Voici un certain nombre de fonctions utiles qui manipulent les chaînes de caractères :

- `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($l);` ;
- `chomp($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. Ne pas écrire `*incorrect* $x=chomp($x) *incorrect*`, car `chomp` renvoie le nombre de caractères supprimés. Cette fonction nous sera très utile lorsque nous lirons des fichiers ligne à ligne ;
- `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`. On rencontrera aussi cette fonction chez les listes (son comportement dépend du contexte) ;
- `substr($x,`offset`,`length`)` vaut la sous-chaîne de position offset et de longueur length. Les positions commencent à 0 :
  `substr("bonjour",1,2)` vaut `on`. La longueur peut être omise, dans ce cas toute la partie droite de la chaîne est sélectionnée.
  Cette fonction peut être une lvalue, c'est-à-dire qu'on peut lui affecter une valeur (lvalue pour left-value : à la gauche du signe égal de l'affectation) :
  ```
  my $v = "salut toi";
  substr($v,5,1) = "ation à ";
  ```
  `$v` vaut alors `"salutation à toi"`. C'est là que l'on se rend compte que Perl gère vraiment la mémoire tout seul !
- `index($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).

En contexte de chaîne de caractères, `undef` vaut la chaîne vide ; le pragma `use warnings;` provoquera un message d'avertissement. Dans ce contexte, un nombre vaut la chaîne de sa représentation décimale.

2.6 Les opérateurs de test

Les booléens (type de données ayant pour seules valeurs vrai et faux) n'existent pas en tant que tels en Perl, on utilise les scalaires pour effectuer les tests (comme C le fait avec les entiers). Il me faut donc préciser quelles sont les valeurs scalaires vraies et quelles sont les fausses.

Les valeurs 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 `''` (ce qui 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...

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 (contexte numérique) et `eq` teste l'égalité de deux chaînes (contexte de chaîne). `("02"=="2")` est vrai alors que `("02" eq "2")` est faux. La différence est encore plus flagrante pour les opérateurs d'infériorité et de supériorité ; `<` 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. Confondre ou mélanger ces deux types d'opérateurs est une erreur *très* courante que font les débutants, ainsi que les initiés qui ne font pas attention... Sachez que le pragma `use warnings;` permet souvent de repérer ces situations.

Voici un tableau décrivant les opérateurs de tests :

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

Les opérateurs booléens classiques sont présents :

- expr1`&&`expr2 est vrai si expr1 et expr2 sont vraies (si expr1 est faux expr2 n'est pas évaluée) ;
- expr1`||`expr2 est vrai si expr1 ou expr2 est vraie (si expr1 est vraie expr2 n'est pas évaluée) ;
- `!`expr est vrai si expr est fausse.

Il existe aussi les opérateurs `and` `or` et `not`. Ceux-ci ont la même table de vérité que les précédents, mais sont d'une priorité plus faible.

Les deux opérateurs cités à la dernière ligne du tableau ne sont pas des opérateurs de test, mais des opérateurs de comparaison ; ils sont présents dans ce tableau en raison des similitudes qu'ils ont avec les opérateurs de test en ce qui concerne le contexte imposé aux opérandes. Ces opérateurs renvoient un nombre qui dépend de l'ordre entre leurs deux paramètres. 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.

Cet opérateur `<=>` est surnommé *spaceship* (vaisseau spatial en français) en raison de sa forme `;-)`... Pour l'opérateur `cmp`, la comparaison se fait sur l'ordre des chaînes selon la table ASCII. Ces opérateurs seront fort utiles lorsque nous parlerons de la fonction `sort` qui effectue le tri des listes.

Chapitre 3

Structures de contrôle

Ici nous allons apprendre à contrôler le flux des instructions en Perl. En effet un programme n'est pas qu'une simple suite d'instructions se déroulant linéairement une fois et une seule.

Il faut savoir que Perl (tout comme le C) permet d'indenter notre code comme bon nous semble, les exemples qui suivent comportent donc des choix personnels d'indentation qui peuvent diverger des vôtres.

3.1 Les instructions de test

Ces instructions permettent de conditionner l'exécution d'instructions à la valeur de vérité d'une expression. L'instruction la plus usitée est le `if` (*si* en français) qui a besoin d'une expression et d'un bloc d'instructions. Cette expression sera évaluée en contexte scalaire et servira de condition ; si elle est vérifiée, le bloc d'instructions sera exécuté.

```
if( $x != 1 ) {
   print "$x\n";
}
```

Ce code Perl a pour effet d'afficher la variable `$x` si elle ne vaut pas 1. Plusieurs instructions peuvent être placées dans le bloc, elles seront alors toutes exécutées si la condition est vraie. Notez que les accolades (`{}`) sont obligatoires pour délimiter le bloc (contrairement au C).

Il est possible d'exécuter d'autres instructions dans le cas où la condition est fausse. On utilise pour cela l'opérateur `else` (*sinon* en français) qui, lui aussi, est suivi d'un bloc d'instructions :

```
if( $x == $y ) {
   print "\$x et \$y sont égaux\n";
} else {
   print "\$x et \$y sont différents\n";
}
```

Le fait que les accolades sont obligatoires a pour conséquence que le programme suivant est incorrect :

```
if( condition1 ) {
   instructions1
```

```
}
else     # Attention ce code est incorrect
   if {
      instructions2
   }
```

Il faut en effet entourer le second `if` par des accolades comme ceci :

```
if( condition1 ) {
   instructions1
} else {
   if( condition2 ) {
      instructions2
   }
}
```

Si le programmeur se sent des envies de devenir sylviculteur en plantant des forêts d'ifs, il faudrait donc qu'il utilise de multiples couples d'accolades. Pour ne pas rencontrer les mêmes problèmes que le Lisp en rencontre pour les parenthèses `;-)`, Perl met à notre disposition l'instruction `elsif` qui permet de cumuler le comportement d'un `else` et d'un `if` tout en faisant l'économie d'un couple d'accolades :

```
if( condition1 ) {
   instructions1
} elsif( condition2 ) {
   instructions2
} else {
   instructions3
}
```

L'instruction `switch` de C n'a pas d'équivalent direct en Perl ; il faut pour cela planter une forêt d'ifs, comme dans l'exemple précédent.

Mais Perl n'en reste pas là. Il existe une syntaxe très utilisée pour effectuer une unique instruction si une condition est vérifiée :

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

On parle ici de *modificateur d'instruction*. Pour cette syntaxe, les parenthèses sont optionnelles autour de la condition, mais je vous conseille de les mettre systématiquement pour une meilleure lisibilité. Le code suivant affiche la variable `$s` si elle est définie :

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

On notera que cette syntaxe ne permet pas l'usage d'un `else`.

L'instruction `unless` a exactement le même rôle que le `if`, à la différence que les instructions seront effectuées si la condition est fausse (il est aussi moins dépaysant d'en faire des forêts). `unless( expression )` est équivalent à `if( !(expression) )` dans toutes les constructions précédemment citées.

3.2 Les boucles

Les boucles permettent d'exécuter plusieurs fois les mêmes instructions sans avoir à écrire plusieurs fois les mêmes lignes de code. Bien souvent nous avons besoin de modifier une variable à chaque étape ; dans ce cas nous utiliserons l'instruction `for` (*pour* en français) dont voici la syntaxe :

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

La boucle `for` prend trois expressions entre parenthèses : la première expression permet d'initialiser la variable de boucle, la deuxième est une condition de continuation et la dernière permet de modifier la valeur de la variable de boucle.

Quand la boucle démarre, la variable est initialisée (expression 1) et le test est effectué (expression 2). Si cette condition est vérifiée, le bloc d'instructions est exécuté. Quand le bloc se termine, la variable est modifiée (expression 3) et le test est de nouveau effectué (expression 2). Si la condition est vérifiée, le bloc d'instructions est réexécuté avec la nouvelle valeur pour la variable de boucle.

Tant que le test reste vrai, le bloc d'instructions et l'expression de modification de la variable sont exécutés. À l'arrêt de la boucle, les instructions qui suivent la boucle sont exécutées.

L'exemple suivant affiche tous les entiers pairs de 0 à 20 inclus :

```
for( my $i=0; $i<=20; $i+=2 ) {
   print "$i\n";
}
```

La boucle s'arrête lorsque `$i` vaut 22. Cette variable est déclarée dans le bloc d'initialisation et n'existe donc que dans la boucle. Notez qu'il est tout à fait possible d'utiliser une variable préexistante comme variable de boucle (et donc de ne pas faire de `my` dans la partie initialisation) ; dans ce cas, après exécution de la boucle, la variable vaut la dernière valeur qui lui a été affectée au cours de la boucle.

Une autre boucle existe : la boucle `while` (*tant que* en français) dont voici la syntaxe :

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

Les instructions sont effectuées tant que la condition est vraie. La partie initialisation doit avoir été effectuée avant la boucle ; la partie modification de la variable doit avoir lieu dans le bloc d'instructions.

L'exemple suivant affiche lui aussi les entiers pairs de 0 à 20 :

```
my $i = 0;
while( $i <= 20 ) {
   print "$i\n";
   $i+=2;
}
```

La seule différence entre les deux exemples est le fait que, dans le cas du `while`, la variable `$i` existe après la boucle.

Comme pour le `if`, certaines facilités sont offertes pour le `while`. Tout d'abord, la syntaxe suivante est correcte :

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

Elle permet d'exécuter plusieurs fois une instruction et une seule tant qu'une condition est vérifiée.

Ensuite, il existe une instruction `until` (*jusqu'à* en français) qui a la même syntaxe que le `while`, mais qui demande une condition d'arrêt (comme `unless` pour `if`) : `until(condition)` est équivalent à `while(!(condition))`.

Lors de l'exécution d'une boucle, il est fréquent de rencontrer des cas particuliers que l'on souhaiterait sauter ou pour lesquels on aimerait mettre fin à la boucle. Les instructions `next`, `last` et `redo` vont nous servir à cela dans les boucles `for`, `while` ou `until`.

L'instruction `next` (*suivant* en français) 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` (*dernier* en français) provoque la fin de la boucle, ni l'incrément ni le test ne sont effectués.

L'instruction `redo` (*refaire* en français) provoque le redémarrage du bloc d'instructions sans que la condition ni l'incrémentation ne soient effectuées.

L'exemple suivant est une autre façon d'imprimer à l'écran les entiers pairs de 0 à 20 :

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

Dans le cas où l'on souhaite exécuter le bloc d'instructions une fois avant d'effectuer le test, on peut utiliser la syntaxe suivante :

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

Pour des raisons trop longues à exposer ici, il ne faut pas utiliser les instructions `next`, `last` et `redo` dans le cas de la boucle `do while`.

Nous verrons dans la suite qu'il existe une autre structure de boucle : `foreach`. Elle permet d'itérer sur les éléments d'une liste (notion que nous aborderons à cette occasion). Vous verrez alors qu'en Perl on utilise beaucoup le `foreach` et assez peu le `for(;;)`.

3.3 Un exemple

Voici un petit exemple de programme Perl ; il n'est pas très utile dans la vie de tous les jours, mais il utilise beaucoup des notions abordées jusqu'ici. Si vous parvenez à comprendre tout ce qu'il fait, vous n'aurez pas perdu votre temps à le lire !


```
 1: #!/usr/bin/perl
 2: use strict;
 3: use warnings;
 4: my $v ="##########";
 5: for( my $i=9; $i>0; $i-- ) {
 6:   print("$i impair\n")
 7:     if( $i % 2 );
 8:   print( "-"x$i . "\n")
 9:     unless( $i % 3 );
10:   substr( $v, $i, 0 ) = $i;
11: }
12: print("$v\n");
```

Voici le détail du rôle des lignes.

- 1 : Le shebang Perl.
- 2 : Cette instruction rend le langage moins permissif, je vous conseille de toujours la placer au début de vos programmes.
- 3 : Cette instruction déclenche d'éventuels messages d'avertissement.
- 4 : Nous déclarons et initialisons une variable scalaire.
- 5 : Une boucle `for`. Déclaration et initialisation à 9 de la variable de boucle `$i`. Nous continuerons tant qu'elle est strictement positive ; à chaque étape nous la décrémenterons : elle variera donc de 9 à 1.
- 6 et 7 : Affichage d'une chaîne dans le cas où `$i` est impair.
- 8 et 9 : Affichage d'une chaîne dans le cas où `$i` est multiple de 3. Cette chaîne comporte `$i` caractères moins (`-`).
- 10 : Insertion de `$i` dans la chaîne `$v` en position `$i` (une longueur de 0 provoque une insertion et non un remplacement).
- 11 : Accolade délimitant le bloc de la boucle.
- 12 : Affichage de la variable `$v`

L'affichage suivant est donc effectué :

```
9 impair
---------
7 impair
------
5 impair
3 impair
---
1 impair
#1#2#3#4#5#6#7#8#9#
```

Avez-vous tout compris ?

Chapitre 4

Listes et tableaux

Les scalaires et les expressions de base n'ont maintenant plus aucun secret pour vous. Des notions plus complexes et des fonctions plus puissantes sont alors à notre portée. C'est le cas des listes, des tableaux et de l'impressionnant arsenal de fonctions Perl permettant de les manipuler ; vous verrez tout ce qu'il est possible de faire en Perl avec ces deux concepts a priori anodins.

Une liste est une suite (donc ordonnée) de valeurs scalaires. Nous verrons comment créer une liste, la manipuler, la parcourir, etc.

Une variable de type tableau peut contenir plusieurs valeurs scalaires. Cette notion est présente dans de nombreux langages de programmation et ne posera sans doute problème à personne.

Les passerelles entre listes et tableaux sont nombreuses et très intuitives en Perl. C'est pour cela que nous n'entrerons pas ici dans les détails de la distinction entre liste et tableau. Dans ce document, j'utiliserai chacun des deux termes à bon escient sans forcement indiquer explicitement pourquoi j'utilise l'un plutôt que l'autre, mais les notions pourront apparaître naturelles au lecteur sans qu'il ne soit nécessaire de préciser les choses formellement.

4.1 Valeurs de listes

En Perl, une liste peut être représentée par les valeurs qu'elle doit contenir encadrées par un couple de parenthèses. Par exemple `(2,5,-3)` est une liste de trois scalaires : 2, 5 et -3. Autre exemple `(2,'age',"Bonjour $prenom")` est aussi une liste ; en effet les listes contenant des scalaires, rien ne nous empêche d'en constituer une comportant des nombres et des chaînes de caractères mêlés. La liste vide se représente sous la forme suivante : `()`

L'opérateur d'intervalle `..` permet de créer une liste comportant des valeurs successives entre deux bornes. La liste `(1..10)` comporte tous les entiers de 1 à 10 ; on aurait pu aussi écrire `(1,2,3,4,5,6,7,8,9,10)`, mais cette dernière notation est bien plus lourde. Il faut savoir que les valeurs des bornes ne doivent pas obligatoirement être des nombres : par exemple, la liste `('a'..'z')` comporte toutes les lettres de l'alphabet, en minuscules et dans l'ordre. Il est aussi possible de spécifier les bornes à l'aide de variables : `($debut..$fin)` On comprendra qu'il n'est pas toujours possible de résoudre ce type de liste (par exemple si `$debut` vaut 1 et `$fin` vaut `'a'`), dans ce cas la liste est vide. Dernier exemple, la liste `(1..10, "age", "a".."z")` comporte 37 éléments (10+1+26).

La liste `(1,2,("nom",12),"aaa",-1)` n'est pas une liste à cinq éléments dont le troisième serait une autre liste, c'est en fait une liste à six éléments. On aurait pu écrire `(1,2,"nom",12,"aaa",-1)`

et on aurait obtenu la même liste. On appelle cela l'aplatissement (ou la linéarisation) des listes. Pour constituer une liste de listes, il faudra faire usage de références (notion que nous aborderons plus tard).

L'opérateur de répétition (`x`), que l'on a déjà appliqué aux chaînes de caractères précédemment, s'applique aussi aux listes : `(2,10) x 3` est une liste à six éléments valant `(2,10,2,10,2,10)`.

4.2 Manipulation de tableaux

Pour simplifier les choses, un tableau est une variable qui peut avoir une liste pour valeur. Une telle variable se déclare de la sorte : `my @t;` On a alors un tableau vide, c'est-à-dire sans élément. De manière plus explicite, voici comment déclarer un tableau vide :

```
my @t = ();
```

Pour lui donner une valeur lors de sa déclaration, il faut faire ainsi :

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

On a alors déclaré ce tableau en l'initialisant au moyen d'une liste.

On peut accéder directement à un élément d'un tableau grâce à son indice : `$t[`*indice*`]` représente l'élément d'indice *indice* du tableau `@t`. Notez bien que la globalité du tableau se représente au moyen d'une arobase `@t` alors qu'un élément particulier est désigné à l'aide d'un dollar `$t[`*indice*`]`, cette dernière expression étant bien une variable de type scalaire (le dollar est réservé aux scalaires en Perl).

Les indices des tableaux en Perl commencent à 0 (comme en C), ce qui signifie que le premier élément du tableau `@t` est `$t[0]` et le deuxième `$t[1]`, etc. Voici un petit exemple d'utilisation de tableau :

```
my @t = (3,5);   # déclaration et initialisation
$t[1] = 4;       # affectation d'un élément
print "$t[0]\n"; # affichage d'un élément
```

Il est intéressant de savoir qu'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]` (ce dernier étant bien plus lisible). Il peut être utile de savoir que l'expression `scalar(@t)` (c'est-à-dire l'utilisation d'un tableau en contexte scalaire) donne le nombre d'éléments du tableau `@t` (ce qui vaut 1 de plus que `$#t`) ; `$x=@t` donnerait la même chose.

Il faut savoir que vous ne générerez pas d'erreur (débordement ou autre) si vous tentez 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` (que l'on retrouvera pour les tables de hachage) permet de tester l'existence d'un élément d'un tableau :

```
if( exists( $t[100] ) ) {
   ...
}
```


Ce test sera vrai si l'élément d'indice 100 du tableau `@t` existe. Ce qui est différent du test suivant :

```
if( defined( $t[100] ) ) {
   ...
}
```

Car on teste ici si l'expression `$t[100]` vaut `undef` ou non, ce qui peut être vrai dans deux cas : soit l'élément existe et vaut `undef`, soit l'élément n'existe pas...

Voici une autre illustration du fait que vous n'avez pas à vous soucier de problèmes d'allocation mémoire :

```
my @t = (3,23.4,"as");
$t[1000] = 8;
```

Ce programme est correct et fonctionne parfaitement : l'affectation à l'indice 1000 agrandit le tableau d'autant... Les éléments d'indice compris entre 3 et 999 valent `undef` et `scalar(@t)` vaut 1001. C'est si facile finalement !

Un tableau qu'il est utile de connaître est `@ARGV`. Cette variable spéciale est toujours définie (même dans les fonctions) et ne nécessite pas de déclaration. Elle contient les arguments de la ligne de commande du programme. Les trois façons de lancer un programme en Perl sont susceptibles d'utiliser `@ARGV` :

```
perl -e '... code perl ...' arg1 arg2 arg3
perl script.pl arg1 arg2 arg3
./script.pl arg1 arg2 arg3
```

Ces trois programmes sont lancés avec les trois mêmes arguments. Sachez que, contrairement au langage C, le nom du programme n'est pas contenu dans `@ARGV` qui ne comporte donc que les arguments au sens strict. La variable spéciale `$0` (comme en shell) contient le nom du programme (nul besoin de déclarer cette variable pour l'utiliser).

4.3 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` : on obtient bien deux tableaux tout à fait identiques (et distincts, la modification de l'un n'entraînant nullement la modification de l'autre).

Voici d'autres instructions d'affectation 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) = (1,2,3);` Les mêmes affectations sont effectuées ici, la valeur 3 n'étant d'aucune utilité ;

- `($a,$b) = (1);` L'affectation à `$a` de la valeur 1 est effectuée et `$b` est mis à `undef` (son ancienne valeur est perdue) ;
- `($a,$b) = @t;` Les variables citées à gauche 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` il 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 ;
- `($a,$b) = Fonction();` Nous verrons un peu plus loin comment écrire une fonction, et comment lui faire renvoyer une liste : ici l'affectation se fait dans les mêmes conditions que pour les trois premiers cas ;
- `($a,$b) = ($b,$a);` Cette instruction est la plus savoureuse : on peut échanger deux variables Perl sans avoir à en utiliser une troisième... (Ai-je déjà dit que Perl s'occupe lui-même de la mémoire ?).

4.4 Multi-déclaration

Pour déclarer plusieurs variables avec un seul `my`, le débutant aurait tendance à écrire la chose suivante (il n'y a pas de honte !) :

```
my $a,$b;  # Incorrect !
```

Ceci est incorrect. Pour pouvoir faire cela, il nous faut utiliser une liste :

```
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.

4.5 Retour sur l'aplatissement des listes

On retrouve la notion d'aplatissement des listes avec les tableaux :

```
@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 contient les cinq éléments, résultat de l'aplatissement du tableau `@t` dans la liste de droite lors de l'affectation de `@t2`. Cette affectation a eu le même résultat qu'aurait eu la suivante :

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

4.6 Absorption d'une liste par un tableau

La syntaxe suivante est intéressante à connaître :

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

Le membre gauche de l'affectation est constitué d'une liste comportant une variable scalaire et un tableau. Il n'y a pas à proprement parler d'aplatissement de liste, car il s'agit ici d'une l-value (membre gauche d'une affectation), mais la variable `$a` reçoit le premier élément du tableau `@s` et le tableau `@t` *absorbe* tous les autres (`@s` n'étant bien sûr pas modifié).

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 (cas idiot, mais bon...) seraient mises à `undef` s'il s'agit de scalaires et à vide s'il s'agit de tableaux. Par exemple, l'affectation suivante :

```
@s = (10,1,2,"age",20);
($a, @t, @u, $b) = @s;
```

équivaut à :

```
@s = (10,1,2,"age",20);
$a = 10;
@t = (1,2,"age",20);
@u = ();
$b = undef;
```

Simple et intuitif.

4.7 La structure de boucle `foreach`

Cette instruction permet de parcourir une liste. Son implémentation optimisée dans l'interpréteur Perl rend son usage bien plus efficace qu'un parcours qui utiliserait une variable indicielle incrémentée à chaque tour d'une boucle `for`. Sa syntaxe est la suivante :

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

À chaque tour de boucle, la variable aura pour valeur un élément de la liste, la liste étant parcourue dans l'ordre. Aucune modification ni suppression dans la liste n'est effectuée par défaut dans ce type de boucle. Il vous est possible de modifier la variable de boucle (ce qui aura pour effet de modifier l'élément en question), mais, par défaut, le parcours n'est pas destructif.

Par exemple :

```
foreach $v (1,43,"toto") {
   print "$v\n";
}
```

Ce petit programme affiche chaque élément de la liste sur une ligne. Ces autres exemples sont valides eux aussi :

```
foreach $v (@t) { .... }
foreach $v (32,@t,"age",@t2) { .... }
```

Dans le premier cas, les éléments du tableau `@t` sont parcourus. Le second exemple illustre les phénomènes d'aplatissement des listes qui se retrouvent ici aussi.

Il est possible de déclarer la variable de boucle dans le `foreach` de la manière suivante :

```
foreach my $v (@t) {
   print "$v\n";
}
```

Il est aussi possible de ne pas utiliser explicitement de variable de boucle ; dans ce cas c'est la variable spéciale `$_` qui sera automatiquement utilisée :

```
foreach (@t) {
   print "$_\n";
}
```

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

Voici un petit exemple d'utilisation de `foreach` affichant des tables de multiplication :

```
#!/usr/bin/perl
use strict;
use warnings;
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 );
   }
   print "\n";
}
```

Et le voici à l'œuvre :

```
./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
```

Passons à la suite.

4.8 Fonctions de manipulation de tableaux

Il existe de nombreuses fonctions permettant de manipuler les tableaux. Pour chacun des exemples qui vont suivre, je suppose que nous avons un tableau `@t` déclaré de la sorte :


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

- Ajout et suppression à gauche
  - La fonction `unshift` prend en arguments un tableau et une liste de valeurs scalaires ; ces valeurs sont ajoutées au début du tableau :
    `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 :
    `$v = shift(@t);`
    `$v` vaut alors 1 et `@t` la liste (2,3,4).
- Ajout et suppression 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 :
    `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 :
    `$v = pop(@t);`
    `$v` vaut alors 4 et `@t` la liste (1,2,3).
- Inversion
  En contexte de liste, la fonction `reverse` prend en argument une liste et renvoie la liste inversée, c'est-à-dire celle dont les éléments sont pris dans le sens opposé :
  `@s = reverse(@t);`
  `@s` vaut alors la liste (4,3,2,1) et `@t` n'est pas modifiée.

Avec de telles fonctions, il est alors envisageable de manipuler des objets algorithmiques tels que les piles et les files. Une pile est un lieu de stockage ayant pour particularité que le dernier élément à y être entré sera le premier à en sortir (last in-first out) comme pour une pile d'assiettes sur une étagère de placard. On peut utiliser pour cela un tableau, avec les fonctions `push` pour ajouter un élément et `pop` pour en prendre un. De façon analogue, une file est un endroit où le premier entré est le premier à sortir (first in-first out) comme pour une file à une caisse de magasin. On peut par exemple utiliser les fonctions `push` pour ajouter un élément et `shift` pour en prendre un.

D'autres manipulations plus complexes du contenu d'un tableau sont possibles avec la fonction `splice`, mais je vous renvoie à la documentation pour les détails.

4.9 L'opérateur `qw`

L'opérateur `qw` nous permet de créer facilement une liste de chaînes de caractères. En effet, il peut sembler pénible de constituer une longue liste de tels éléments en raison du fait qu'il faut délimiter chacun d'entre eux au moyen de simples ou de doubles-quotes :

```
@t = ( 'Ceci', 'est', 'quelque', 'peu', 'pénible',
       'à', 'écrire', ',', 'non', '?' );
```

Avec `qw`, ceci devient tout à coup plus lisible :

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

La chaîne de caractères sera découpée selon les espaces, les tabulations et les retours à la ligne.

Les délimiteurs les plus souvent utilisés sont les parenthèses (comme dans l'exemple précédent) ainsi que les slashs :

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

Cette fonction est bien pratique, mais peut être source d'erreurs, voyez l'exemple suivant :

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

Les simples quotes (') semblent indiquer que le programmeur souhaite constituer un seul élément comportant les mots `aux` et `erreurs` ; ce n'est pas ce qui est fait ici. En effet, ni les simples quotes ni les doubles-quotes ne constituent un moyen de regrouper des mots pour l'opérateur `qw`. La liste ainsi créée comporte donc quatre éléments ; on aurait pu écrire :
`("attention","'aux","erreurs'","bêtes")`.

4.10 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. Les arguments passés ne sont pas modifiés.

scalaire `= join(` *séparateur*`,` *liste* `);`

Voici quelques 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 vaut alors la concaténation des valeurs du tableau `@t` avec `" : "` pour séparateur.

4.11 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 le séparateur. Le séparateur est une expression régulière, notion que nous aborderons dans la suite, mais dont le minimum de connaissances suffit à cette fonction ; admettez ici qu'une telle expression est à placer entre slashs (`/`). Les arguments passés ne sont pas modifiés.

liste `= split(` `/`*séparateur*`/,` *chaîne* `);`

Voici quelques exemples :

- `@t = split(/-/,"4-12-455");` Le tableau comporte alors les éléments 4, 12 et 455.
- `($x,$y) = split(/==/,$v);` Les deux variables auront pour valeur les deux premières chaînes de caractères qui soient séparées par deux signes d'égalité.
- `print join(':',split(/ /,'salut ici'));` Affiche `salut:ici` (il existe des méthodes plus efficaces et plus lisibles de faire cela...).

4.12 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* `);` (attention à ne pas mettre de virgule entre le bloc d'instructions et la liste).

Tout au long du tri, le bloc d'instructions sera évalué pour comparer deux valeurs de la liste ; ces deux valeurs sont localement affectées aux variables spéciales `$a` et `$b` qui ne sont définies que dans le bloc et sur lesquelles il faut donc effectuer la comparaison. Il faut faire particulièrement attention au fait que s'il existe des variables `$a` et `$b` dans le programme elles seront localement masquées par ces variables spéciales (source courante d'erreurs). Le bloc doit être composé d'une expression dont la valeur est :

- négative, si `$a` doit être avant `$b` dans la liste résultat ;
- positive, si `$b` doit être avant `$a` ;
- nulle, s'ils sont équivalents.

C'est là qu'entrent en jeu les opérateurs de comparaison `cmp` et `<=>` : ils 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.

Voici quelques exemples :

- `@s = sort( {$a cmp $b} @t );` La liste `@s` a pour valeur la liste `@t` triée selon l'ordre lexical.
- `@s = sort( @t );` Le fonctionnement est identique à l'exemple précédent.
- `@s = sort( {$a <=> $b} @t );` Le critère de tri est ici numérique.
- `@s = sort( {length($b) <=> length($a) or $a cmp $b} @t );` Une expression composée peut bien sûr servir de critère : le tri est ici d'abord numérique inverse sur la longueur puis lexical. Cela permet d'effectuer un second tri pour les éléments égaux selon le critère du premier tri.
- `@s = sort( { fonction($a,$b) } @t );` Vous pouvez écrire votre propre fonction de tri (à deux arguments) ; elle doit renvoyer un nombre dont la valeur dépend de l'ordre voulu (voir juste avant).

4.13 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 (cas sur lequel nous reviendrons plus tard), soit un bloc d'instructions (cas sur lequel nous allons nous étendre) :

liste2 `= grep {` *sélection* `}` *liste1*`;` (attention : pas de parenthèses ni de virgule).

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 donc être effectués.

Voici quelques exemples :

- `@t = grep { $_<0 } $x,$y,$z;` Affecte à `@t` les éléments négatifs de la liste.

- `@s = grep { $_!=8 and $_!=4 } @t;` Met dans `@s` les éléments de `@t` différents de 4 et de 8.
- `@s = grep { fonction($_) } @t;` Vous pouvez écrire votre propre fonction de sélection ; elle doit renvoyer vrai ou faux selon que l'élément est à garder ou non.

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* `);`

En quelques mots, les éléments renvoyés seront ceux qui correspondront à l'expression régulière. Par 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 seront données dans la suite.

J'ai affirmé que la liste d'origine n'était pas modifiée, mais il vous est possible de le faire. Si, durant la sélection, vous affectez une valeur à `$_`, la liste sera modifiée. Mais cela est sans doute une mauvaise idée de modifier la liste passée en paramètre d'un `grep`, car la fonction `map` est faite pour cela.

4.14 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* `);` (attention à ne pas mettre de virgule entre le bloc d'instructions et la liste).

La variable spéciale `$_` vaut localement (dans le bloc d'instructions) chaque élément de la liste. La valeur de la dernière expression du bloc sera placée dans la liste résultat.

Voici quelques exemples :

- `@s = map( { -$_ } @t );` Le tableau `@s` aura pour valeurs les opposés des valeurs de `@t`.
- `@p = map( { $_."s" } @t );` Tous les mots de `@t` sont mis au pluriel dans `@p`.
- `@s = map( { substr($_,0,2) } @t );` Le tableau `@s` aura pour valeurs les deux premiers caractères des valeurs de `@t`.
- `@s = map( { fonction($_) } @t );` Vous pouvez écrire votre propre fonction ; les valeurs qu'elle renverra seront placées dans `@s`.

Dans les exemples qui précèdent, la liste d'origine n'est pas modifiée (sauf dans le dernier exemple où elle peut l'être dans la fonction). Voici un exemple de modification de liste :

`map( { $_*=4 } @t );` Tous les éléments de `@t` sont multipliés par quatre.

Chapitre 5

Écrire une fonction

Une fonction est un ensemble d'instructions regroupées de manière à être utilisées plusieurs fois sans avoir à dupliquer du code.

5.1 Déclaration

Le mot clef `sub` permet de définir des fonctions en Perl. Les arguments d'une fonction sont des valeurs scalaires, à l'exclusion de toutes autres (on verra comment faire en sorte de passer un tableau en argument) ; ces paramètres sont accessibles via la variable spéciale `@_` (qui est donc un tableau). Modifier une valeur de `@_` modifiera les variables d'appel, il est donc d'usage d'en faire une copie avant manipulation.

```
sub maJolieFonction {
   my ($x,$y,$t) = @_;
   ... instructions ...
   return $z;
}
```

Ces quelques lignes définissent une nouvelle fonction dont le nom est `maJolieFonction`. Cette fonction copie dans trois variables locales les trois premières valeurs du tableau `@_`, c'est-à-dire ses trois premiers paramètres (les règles classiques d'affectation entre listes et tableaux s'appliquent ici). Je vous conseille de toujours commencer vos fonctions par une ligne copiant les valeurs de `@_` et de ne plus utiliser `@_` dans la suite de la fonction (sauf cas spécial). Si votre fonction attend un seul paramètre, la syntaxe peut être la suivante :

```
my ($x) = @_;
```

mais ne peut pas être :

```
my $x = @_; #incorrect
```

Cette ligne est incorrecte, car dans ce cas, la variable `$x` aurait pour valeur le nombre de paramètres (affectation d'un tableau à un scalaire). La syntaxe suivante peut aussi être utile :

```
my ($x,@t) = @_;
```

la variable `$x` reçoit le premier paramètre et le tableau `@t` reçoit tous les paramètres restants.

Enfin, une autre écriture que vous verrez souvent dans les programmes Perl est la suivante :

```
my $x = shift;
```

celle-ci s'appuie sur le fait que dans une sous-routine, la fonction `shift` travaille par défaut sur `@_`.

L'instruction `return` met fin à l'exécution de la fonction et on peut lui fournir une expression qui sera alors la valeur de retour de la fonction.

5.2 Appel

La fonction ainsi définie peut être appelée au moyen de la syntaxe suivante :

```
maJolieFonction(10,20,30);
```

Dans ce cas, l'éventuelle valeur de retour est ignorée. Pour récupérer cette valeur :

```
$v = maJolieFonction(10,20,30);
```

Il est possible d'omettre les parenthèses lors de l'appel à une fonction :

```
maJolieFonction 10,20,30;  # À éviter
```

mais cela peut créer des ambiguïtés et je vous déconseille donc cette syntaxe.

S'il est possible en Perl d'imposer le nombre d'arguments pour une fonction (nous n'en parlerons pas ici), cela n'est pas fait par défaut. Rien ne nous empêche en effet d'appeler la fonction `maJolieFonction` précédemment définie avec deux ou quatre arguments, alors qu'elle semble en attendre trois ; si on l'appelle avec deux arguments, la variable `$t` vaudra `undef` ; par contre si on l'appelle avec plus de trois arguments, les valeurs suivantes seront ignorées. Mais cette particularité du langage est parfois bien pratique, notamment pour écrire des fonctions à nombre variable d'arguments.

5.3 Visibilité des variables

Les variables déclarées au moyen de `my` dans une fonction ne seront visibles qu'à l'intérieur même de la fonction, dans le code qui suit la déclaration. Dans une fonction, il est possible d'accéder aux variables définies à la « racine » du programme (c'est-à-dire en dehors de toute fonction) : il s'agit donc 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 maJolieFonction {
   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
}
maJolieFonction();
print "$a\n";        # affiche 3
print "$b\n";        # affiche 8
print "$c\n";        # affiche 15
```

De manière plus générale, les variables déclarées au moyen de `my` sont visibles jusqu'à la fin du plus petit bloc qui les englobe. En particulier, dans une fonction...

```
sub maJolieFonction2 {
   my $d = -3;
   if( ... ) {
      my $d = 4;
      my $e = 8;
      print "$d\n";  # affiche 4
      print "$e\n";  # affiche 8
   }
   print "$d\n";     # affiche -3
   print "$e\n";     # $e n'existe pas ici
}
```

5.4 Une liste pour valeur de retour

Il est tout à fait possible de faire renvoyer plusieurs scalaires par une fonction, il suffit d'utiliser une liste. Voici des exemples de syntaxe de liste renvoyée par des fonctions :

```
   return ($x,$z);
   return @t;
```

Dans le second cas, le tableau est converti en liste. Et voici comment il est possible de récupérer ces valeurs :

```
@s = fonction(...);
($j,$k) = fonction(...);
```

Ces deux manières de procéder peuvent parfaitement être utilisées chacune dans les deux cas de `return` pré-cités (ce sont toujours les mêmes règles d'affectation qui s'appliquent).

5.5 Premier exemple de fonction

Voici un exemple complet de programme en Perl avec une fonction :

```
 1:  #!/usr/bin/perl
 2:  use strict;
 3:  use warnings;
 4:  my $t = "Bonjour Larry";  # variable globale
 5:  print "$t\n";             # avec ou sans parenthèses
 6:  sub f {
 7:     my ($x,$z) = @_;       # deux arguments attendus
 8:     my $m = $x*$z;
 9:     printf("%d\n", $m);
10:     return ($x+$z,$m);     # retourne une liste
11:  }
12:  my @t = f(3,5);
13:  print "$t $t[0] $t[1]\n";
```

Un bon programme Perl commence toujours par les première et deuxième lignes. Si la variable scalaire `$t`, elle, est globale, en revanche les variables `$x`, `$z` et `$m` sont locales à la fonction. En ligne 12, le tableau `@t` reçoit pour valeur la liste renvoyée par la fonction. Notez bien qu'il n'y a aucun conflit entre les variables `$t` et `@t` ; en effet, l'instruction de la dernière ligne procède d'abord à l'affichage de la variable scalaire `$t` puis du premier et deuxième éléments du tableau `@t` (les crochets permettent de savoir qu'il s'agit d'éléments d'un tableau).

5.6 Autre exemple : une fonction récursive

Voici un exemple de fonction. Elle est récursive (c'est-à-dire qu'elle fait appel à elle-même) : nous allons calculer la factorielle d'un nombre. Par définition, F(0)=F(1)=1 et F(n)=n×F(n-1) pour tout n supérieur à 1 :

```
sub Fact {
   my ($n) = @_;
   return 1
      if( $n == 1 || $n == 0 );
   return $n * Fact($n-1);
}
print Fact(5)."\n";  # affiche 120
```

Aussi lisible que dans tout autre langage.

5.7 Dernier exemple : le crible d'Ératosthène

Nous allons ici illustrer l'usage des listes et des tableaux par un exemple mathématique : le crible d'Ératosthène. Cet algorithme permet de calculer tous les nombres premiers inférieurs à un nombre donné n.

Son principe est le suivant : nous construisons tout d'abord la liste de tous les entiers de 2 à n. Ensuite, à chaque itération, nous supprimons de la liste tous les multiples du premier nombre de la liste et signalons ce premier nombre comme étant premier. Au premier tour de boucle, je supprime tous les nombres pairs et dis que 2 est premier. Au suivant, je supprime tous les

multiples de 3 et affirme que 3 est premier. Au tour suivant, c'est le 5 qui est au début de la liste (4 étant multiple de 2, il a déjà été supprimé), j'enlève de la liste les multiples de 5 et annonce la primalité de 5, etc. L'algorithme se termine lorsque la liste est vide, j'ai alors déterminé tous les nombres premiers inférieurs à n.

Voici la fonction réalisant cet algorithme en Perl :

```
sub Crible {
   my ($n) = @_;

   # Liste initiale :
   my @nombres = (2..$n);
   # Liste des nombres premiers trouvés :
   my @premiers = ();

   # Tant qu'il y a des éléments (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
      @nombres = grep { ( $_ % $prem )!=0 } @nombres;
   }

   # On renvoie la liste des nombres premiers
   return @premiers;
}
```

Quiconque a déjà réalisé cet algorithme en C ou C++ comprendra la joie que cette concision procure...

Chapitre 6

Tables de hachage

Les tables de hachage de Perl ne se retrouvent pas dans beaucoup d'autres langages ; pour les avoir souvent utilisées en Perl, il est dur de repasser à des langages qui n'en sont pas pourvus.

Une table de hachage (*hash table* en anglais) est un type de donnée en Perl permettant d'associer une valeur à une clef. On peut dire d'un tableau (notion abordée précédemment) qu'il associe une valeur scalaire à un entier : à la position i (pour i entier), une certaine valeur scalaire est présente. Une table de hachage va nous permettre d'aller au-delà : on pourra faire correspondre une valeur scalaire (comme pour un tableau) à toute chaîne de caractères (plutôt qu'à un entier).

Je peux, par exemple, avoir envie de gérer en Perl un index téléphonique simple : chacun de mes amis a un numéro de téléphone, je veux pouvoir retrouver leur numéro à partir de leur prénom. Je vais donc associer le numéro au prénom :

```
"Paul"     -> "01.23.45.67.89"
"Virginie" -> "06.06.06.06.06"
"Pierre"   -> "heu ..."
```

Les prénoms seront les clefs, c'est-à-dire le "point d'entrée" dans la table de hachage (comme les indices numéraux le sont pour les tableaux). Les numéros de téléphone seront les valeurs associées à ces clefs. Il s'agit bien d'une association chaîne de caractères vers scalaire.

Vous l'avez sans doute compris, dans une table de hachage, une clef n'est présente qu'une seule fois et ne peut donc avoir qu'une seule valeur (comme l'élément d'un indice donné d'un tableau). Par contre, une valeur peut être associée à plusieurs clefs.

6.1 Déclaration et initialisation

Une variable de type table de hachage se déclare de la sorte :

```
my %h;
```

On a alors une table de hachage vide (aucune clef). Il est possible de signaler explicitement que l'on déclare une table de hachage vide :

```
my %h = ();
```

Pour donner des valeurs initiales à notre table de hachage, on peut utiliser la syntaxe suivante :

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

Cette dernière table de hachage est déclarée et initialisée avec les clefs `Paul`, `Virginie` et `Pierre` ayant respectivement pour valeurs
`01.23.45.67.89`, `06.06.06.06.06` et `heu ...`

6.2 Accéder à un élément

Dans une table de hachage `%h`, on peut accéder à la valeur d'une clef au moyen de la syntaxe suivante : `$h{`*clef*`}` ; par exemple `$h{Paul}` vaut `01.23.45.67.89`. Si la clef comporte d'autres caractères que des lettres, des chiffres et le souligné (underscore en anglais '_'), il faut la délimiter au moyen de simples ou de doubles-quotes : `$h{"Marie-Pierre"}` ou
`$h{'Marie-Pierre'}`.

En fait, cette syntaxe force un contexte de chaîne de caractères entre les accolades, ce qui fait qu'un mot simple (*bareword* en anglais) sera converti silencieusement en chaîne de caractères (même en positionnant le pragma `use warnings;`).

De façon similaire aux tableaux avec l'arobase (`@t`), la totalité d'une table de hachage se représente au moyen du signe pourcentage (`%h`), alors qu'une valeur particulière est désignée à l'aide d'un dollar `$h{`*clef*`}`, cette dernière expression étant bien une variable de type scalaire.

Voici quelques exemples de manipulation d'éléments de la table de hachage `%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 ..." ) {
   ...
}
```

La clef utilisée pour cette syntaxe peut tout à fait être contenue dans une variable scalaire (qui sera évaluée en contexte de chaîne de caractères) :

```
my $k = "Jacques";
$h{$k} = "02.02.02.02.02";
```

Elle peut même être une expression plus complexe :

```
sub f { return "Jac"; }
$h{f().'ques'} = "02.02.02.02.02";
```


6.3 Parcours

Il existe trois fonctions permettant de parcourir une table de hachage. Dans les exemples fournis, nous considérerons que la table `%h` a été déclarée ainsi :

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

- `keys` : obtenir une liste des clefs
  Cette fonction 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.
  `my @t = keys(%h);`
  Le tableau `@t` peut par exemple valoir la liste `("Virginie", "Pierre", "Paul")`.
  Cette fonction va nous permettre de parcourir toute la table de hachage en effectuant une boucle sur la liste des clefs :
  ```
  foreach my $k (keys(%h)) {
     print "Clef=$k Valeur=$h{$k}\n";
  }
  ```
  La variable de boucle `$k` prendra pour valeurs successives l'ensemble des clefs de la table, l'expression `$h{$k}` est la valeur associée à la clef `$k`. Ce petit programme affichera donc tous les couples clef/valeur de la table `%h`.
- `values` : obtenir une liste des valeurs
  De la même façon que `keys` renvoie une liste des clefs d'une table de hachage, la fonction `values` fournit une liste des valeurs ; pour cette fonction non plus, l'ordre n'est pas garanti et seule l'exhaustivité l'est.
  L'exemple suivant
  ```
  foreach my $v (values(%h)) {
     print "Valeur=$v\n";
  }
  ```
  affichera tous les numéros de téléphone (c'est-à-dire les valeurs) de la table `%h`.
  Il n'est bien sûr pas possible de retrouver la clef des valeurs que l'on manipule ainsi.
  Il peut être intéressant de savoir que 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 entretemps.
- `each` : itération sur les couples (clef,valeur)
  Cette fonction 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, c'est-à-dire qu'elle ne renvoie pas toujours la même chose d'un appel à l'autre : en effet, elle renvoie le couple suivant ! De ce fait, je vous conseille de toujours l'utiliser dans la syntaxe qui suit :
  ```
  while( my ($k,$v) = each(%h) ) {
     print "Clef=$k Valeur=$v\n";
  }
  ```

Libre à vous de parcourir vos tables de hachage avec la fonction qui vous convient le mieux.

6.4 Autovivification

Sous ce terme barbare se cache une idée simple : si vous tentez de modifier un élément d'une table de hachage qui n'existe pas, il sera créé. S'il est utilisé dans un contexte numérique, il prendra pour valeur initiale zéro. S'il est utilisé dans un contexte de chaîne de caractères, il prendra pour valeur la chaîne vide (depuis Perl 5.6).

Par exemple, considérons une table de hachage qui ne comporte pas la clef `hello` ; l'expression suivante

```
$h{hello} .= "après";
```

associe à la clef `hello` la valeur chaîne vide puis lui concatène la chaîne `"après"`. De la même façon, l'expression

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

crée un élément de 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. Par exemple nous allons pouvoir compter le nombre d'occurrences des mots dans un texte de manière très simple. Supposons que les mots du texte soient déjà dans un tableau (par exemple en utilisant la fonction `qw` ; elle ne règle pas les problèmes des ponctuations, des majuscules et des lettres accentuées, mais elle suffira à notre exemple). Nous allons utiliser chaque mot comme une clef de la table et nous allons ajouter 1 à la valeur de cette clef :

```
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";
}
```

Ce qui donne l'affichage suivant :

```
Le mot 'vous' est présent 1 fois
Le mot 'bonjour' est présent 2 fois
```

Dans la suite nous verrons comment découper un texte en mots au moyen des expressions régulières.

6.5 Existence et suppression d'une clef

À la lecture de ce qui précède, il peut sembler impossible de savoir si un élément d'une table de hachage existe ou non. Rassurez-vous, les auteurs de Perl ont tout prévu :-) L'opérateur `exists` renvoie vrai si l'élément de table de hachage qu'on lui donne en paramètre existe ; sinon il renvoie faux. Par exemple :

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

Il est important de noter qu'un test effectué au moyen de l'opérateur `defined` aurait été possible, mais dangereux. En effet, l'expression `defined( $h{hello} )` est fausse dans deux cas très différents : soit si l'élément n'existe pas, soit si l'élément existe et vaut `undef` ; elle sera vraie si l'élément existe et ne vaut pas `undef`. Il est donc impossible de distinguer le cas d'un élément absent et celui d'un élément indéfini (valant `undef`) avec `defined`.

Cette distinction entre absent et indéfini peut paraître artificielle dans ce cas (elle peut tout de même être importante dans certaines situations !), mais dans le cas de la suppression d'une clef, il en est tout autrement.

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). De la même façon que `exists` est la bonne méthode pour tester l'existence d'un élément, `delete` est la bonne méthode pour en supprimer un. Le débutant pourrait être tenté d'écrire :

```
$h{hello} = undef;  # attention!
```

Ce qui est fort différent, car dans ce cas, la clef `hello` aura une valeur indéfinie, mais existera toujours ! On la retrouvera, par exemple, dans les parcours effectués au moyen des opérateurs `keys`, `values` ou `each` ; ce qui n'est sans doute pas le but recherché.

Pour résumer, on peut dire que pour tester l'existence d'une clef, il faut utiliser `exists` et que pour en supprimer une, il faut utiliser `delete`.

En marge de ces deux fonctions, voici une manière de savoir si une table de hachage est vide ou non (on qualifie de vide une table de hachage qui ne comporte aucune clef). Cette syntaxe utilise la table de hachage en contexte de chaîne de caractères, par exemple de cette façon :

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

La valeur d'un hachage en contexte scalaire n'a pas d'autre utilisation que celle-ci. En effet, `scalar(%A)` renvoie une valeur du type `4/8` qui indique le nombre de places (*buckets* en anglais) utilisées par rapport au nombre total disponible dans le hachage. Une table vide est un cas particulier, elle renverra `0`.

6.6 Tables de hachage et listes

On peut facilement passer d'une liste (ou tableau) à une table de hachage et inversement. Voyez, par exemple, le petit programme suivant :

```
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 six 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 ...");
```

Il est à noter que cette syntaxe rappelle étrangement l'un des premiers exemples de création de table de hachage qui utilisait `=>` pour séparer clefs et valeurs. Cette similarité est en fait une quasi-équivalence, car l'opérateur `=>` peut être utilisé à la place de la virgule pour créer des listes ; il n'a été ajouté au langage Perl que pour faciliter la lecture des affectations de tables de hachage, car il force un contexte de chaîne à sa gauche, ce qui permet justement d'écrire `%a = ( toto => 'titi' );`

La conversion dans l'autre sens 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 entre couples. La table de hachage `%h` de l'exemple précédent peut être affectée à un tableau :

```
my @t2 = %h;
```

Le tableau `@t2` sera initialisé, par exemple, avec la liste suivante :
`("Pierre", "heu ...", "Paul", "01.23.45.67.89", "Virginie", "06.06.06.06.06")` ; chaque clef précède sa valeur, mais l'ordre des couples (clef,valeur) est quelconque (un peu comme pour la fonction `each`).

Une table de hachage se convertit en liste sans encombre dès qu'elle est en contexte de liste. Je vous laisse deviner ce que fait le code suivant :

```
foreach my $x (%h) {
   print "$x\n";
}
```

La fonction `reverse`, qui nous a permis d'inverser les listes, 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 lors de la transformation de table de hachage en liste l'ordre est quelconque...

L'association individu - numéro de téléphone est idéale pour illustrer cela :

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

On pourra alors retrouver la personne à partir de son numéro de téléphone. Si, par contre, Paul et Virginie avaient eu le même numéro, on n'aurait pas pu prédire quelle serait la personne renvoyée.

6.7 Exemples

Voici quelques exemples d'utilisation des tables de hachage.

Le premier concerne la variable spéciale `%ENV` qui contient les variables d'environnement du programme. `$ENV{PATH}` contient le *path*, `$ENV{HOME}` vaut le nom du répertoire personnel de l'utilisateur qui exécute le programme, etc.

Deuxième exemple, les tables de hachage peuvent servir à constituer des tableaux à plusieurs dimensions ; on pourrait en effet imaginer avoir des clefs qui seraient la concaténation des coordonnées dans les n dimensions : *dim1 :dim2 :dim3 :...*

```
my %h = ();
foreach my $i (0..4) {
   foreach my $j (-3..10) {
      foreach my $k (130..148) {
         $h{"$i:$j:$k"} = Calcul($i,$j,$k);
      }
   }
}
```

Nous verrons dans la suite qu'il est possible de bâtir de réels tableaux à plusieurs dimensions en utilisant des références.

L'exemple suivant concerne les ensembles ; nous allons utiliser les tables de hachage pour calculer l'union et l'intersection de deux ensembles.

```
# Voici mes deux ensembles
# Je mets les éléments dans des tableaux
my @ensA = (1, 3, 5, 6, 7, 8);
my @ensB = (2, 3, 5, 7, 9);

# Voici mon union et mon intersection,
# les éléments des ensembles en seront les clefs
my %union = ();
my %inter = ();

# Je mets tous les éléments de A dans l'union :
foreach my $e (@ensA) { $union{$e} = 1; }

# Pour tous les éléments de B :
foreach my $e (@ensB) {

    # S'il est déjà dans l'union, c'est qu'il est
    # dans A : je le mets donc dans l'intersection :
    $inter{$e} = 1 if ( exists( $union{$e} ) );
```

```
    # Je le mets dans l'union
    $union{$e} = 1;
}

# Tous les éléments présents dans A ou B
# sont des clefs de la table union.
# Tous les éléments présents dans A et B
# sont des clefs de la table inter.

# Je reconstitue des tableaux à partir
# des tables de hachage (en les triant
# pour l'affichage)
my @union = sort( {$a<=>$b} keys(%union) );
my @inter = sort( {$a<=>$b} keys(%inter) );

print("@union\n");
# affiche : 1 2 3 5 6 7 8 9
print("@inter\n");
# affiche : 3 5 7
```

Pour le même problème, voici une solution n'utilisant qu'une seule table de hachage, je vous laisse le soin d'en apprécier le principe :

```
my @ensA = (1, 3, 5, 6, 7, 8);
my @ensB = (2, 3, 5, 7, 9);

my %hash = ();  # Qu'une seule table ...

foreach my $e (@ensA) { $hash{$e}++; }
foreach my $e (@ensB) { $hash{$e}++; }

my @union = sort( {$a<=>$b} keys(%hash) );
my @inter = sort( {$a<=>$b}
                  ( grep { $hash{$_}==2 } keys(%hash) )
                );

print("@union\n");
# affiche : 1 2 3 5 6 7 8 9
print("@inter\n");
# affiche : 3 5 7
```

La compréhension de cet exemple demande d'avoir assimilé plusieurs notions importantes vues jusqu'ici.

6.8 Tranches de tableau

Maintenant que nous connaissons tous les types de données de Perl, notamment les tableaux et les tables de hachage, nous allons voir comment on peut manipuler plusieurs éléments d'un tableau ou d'une table de hachage à la fois. Cela s'appelle une *tranche* (slice en anglais).

Une tranche de tableau est un sous-ensemble des éléments du tableau. Imaginons par exemple un tableau `@t` duquel nous souhaiterions manipuler les éléments d'indice 4 et 10 ; pour cela nous allons prendre la tranche correspondante de ce tableau : `@t[4,10]` est une liste à deux éléments qui est équivalente à `($t[4],$t[10])`. Quelques explications sur la syntaxe. Tout d'abord, l'expression commence par une arobase, car il s'agit d'une liste d'éléments ; le dollar est réservé aux scalaires, par exemple `$t[4]` est un scalaire. Ensuite, comme d'habitude pour les tableaux, les crochets permettent de spécifier les indices. Enfin, l'ensemble des indices est indiqué par une liste d'entiers : `@t[2,10,4,3] @t[3..5] @t[fonction()]`...

Une telle tranche est utilisable comme valeur (passage de paramètres, etc.) et comme l-value (expression à gauche du signe égal d'affectation) :

```
@t[4,10] = (4321,"age");
```

cette instruction affecte 4321 à l'indice 4 du tableau `@t` et la chaîne `age` à l'indice 10. On aurait pu écrire

```
($t[4],$t[10]) = (4321,"age");
```

Une autre utilisation des tranches de tableau apparaît avec les fonctions qui renvoient une liste. Par exemple la fonction `stat` prend en paramètre un nom de fichier et renvoie toutes sortes d'informations sur le fichier : taille, dates, propriétaire etc. Il est courant d'écrire :

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,
 $atime,$mtime,$ctime,$blksize,$blocks) = stat($filename);
```

La fonction renvoie une liste qui est affectée aux variables de la liste de gauche. Les tranches peuvent intervenir si seules quelques informations vous intéressent et que vous ne voulez pas déclarer de variables inutiles. Par exemple, si seules les dates de modification (indice 9) et de création (indice 10) vous intéressent, vous pouvez écrire :

```
($mtime,$ctime) = ( stat($filename) )[9,10];
```

L'appel à la fonction est placé entre parenthèses et on ne prend que les éléments d'indice 9 et 10 de sa valeur de retour. On a alors une liste à deux éléments, celle-ci est affectée à la liste à gauche du signe égal et donc ces deux éléments sont affectés aux deux variables.

6.9 Tranches de table de hachage

De la même façon qu'il existe des tranches pour les tableaux et les listes, il en existe pour les tables de hachage. La sélection s'effectue bien sûr sur les clefs. Par exemple, si `%h` est une table de hachage, alors `@h{'clef1','clef2'}` est une liste équivalente à `($h{'clef1'},$h{'clef2'})` Il est ensuite possible d'utiliser cette liste comme bon vous semble (affectation, passage en paramètre, etc.).

Une utilisation (assez complexe) des tranches serait indiquée lorsque l'on veut construire automatiquement une liste de valeurs uniques à partir d'un tableau dont on n'est pas sûr que ses valeurs soient uniques :

```
# Un tableau avec des valeurs dupliquées :
my @t = qw(hello toto hello vous);

# Déclaration d'une table de hachage :
my %h;

# On prend la tranche de %h dont les clefs
# sont les valeurs du tableau @t
# et on leur associe la valeur undef
@h{@t} = ();

# Les clefs de %h sont donc constituées des
# valeurs de @t, et on est sûr de ne les
# retrouver qu'une seule fois :
@t = keys %h;
```

Le tableau `@t` comporte alors une fois et une seule chacun de ses éléments.

Chapitre 7

Manipulation de fichiers

Pour le moment nous avons écrit des programmes dont les interactions avec leur environnement étaient faibles. Nous allons voir dans cette partie comment manipuler des fichiers en Perl. Les fichiers se retrouvent dans tous les langages, mais la manière très simple et très puissante de les manipuler fait des fichiers une facilité de Perl.

7.1 Opérateurs sur les noms de fichier

Perl dispose d'opérateurs prenant en paramètre un nom de fichier ; ce nom de fichier doit être un scalaire (une variable ou une constante). Leur valeur de retour est souvent booléenne et quelquefois numérique. Les coutumiers du shell retrouveront de nombreuses options de la commande `test`.

- `-e` teste si son paramètre est un chemin valable dans le système de fichiers (répertoire, fichier, etc.). On pourrait l'utiliser ainsi :
  ```
  if( -e "/usr/tmp/fichier" ) {
     print "Le fichier existe\n";
  }
  ```
- `-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. Ceci n'exclut pas que `-f` ou `-d` renvoie vrai.
- `-r` teste si le programme a le droit de lire le fichier/répertoire/etc passé en paramètre.
- `-w` teste si le programme a le droit d'écrire.
- `-x` teste si le programme a le droit d'exécuter le fichier ou d'accéder (ou axéder `:-)`) au répertoire.
- `-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.
- `-M` renvoie l'âge en jour du fichier (depuis le début de l'exécution du programme).

Il existe tout plein d'autres opérateurs sur les fichiers ; pour en connaître la liste complète, je vous invite à lancer la commande `perldoc -f -X`

Voici quelques exemples d'utilisation de ces opérateurs :

```
my $file = "/usr/doc/perl";
```

```
if( -f $file && -w $file ) { .... }
my $taille = -s $file;
my $age = -M $file;
```

Simple et efficace.

7.2 La fonction `glob`

La fonction `glob` prend en argument une expression et renvoie une liste de noms de fichiers. Cette liste correspond à l'évaluation de l'expression selon les *wildcards* du shell.

Par exemple, `glob( '*.o' )` renvoie la liste des fichiers du répertoire courant ayant l'extension `.o`

Notez bien qu'il ne s'agit pas d'une expression régulière (pour ceux qui connaissent ; pour les autres, nous en parlerons plus tard), mais bien d'une expression telle qu'on la donnerait à un shell : `ls [A-Z]*.h` liste tous les fichiers commençant par une majuscule ayant l'extension `.h`

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 d'include pour le C du répertoire `/usr/include`

7.3 Premiers exemples

Voici un premier exemple de manipulation de noms de fichiers :

```
#!/usr/bin/perl
use strict;
use warnings;
foreach my $name ( glob('*') ) {
   print "$name\n"  if( -l $name );
}
```

Il affiche les liens symboliques du répertoire courant.

Voici un second exemple :

```
#!/usr/bin/perl
use strict;
use warnings;
foreach my $name ( glob('.* *') ) {
   next if( ! -d $name );
   next if( ! -w $name );
   print "$name : ". ( -s $name ) ."\n";
}
```

Ce programme affiche le nom et la taille des sous-répertoires du répertoire courant sur lesquels j'ai les droits d'écriture (y compris ceux commençant par un point, donc `.` et `..`).

7.4 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(` *descripteur*`,` *mode*`,` *nom-de-fichier* `)`

Le paramètre *descripteur* sera l'identifiant du fichier après ouverture (on pourrait parler de descripteur de fichier). C'est ce descripteur qui devra être fourni aux fonctions de lecture et d'écriture pour manipuler le fichier. Il s'agit d'une variable que vous devez déclarer, par exemple directement dans le `open`.

Le paramètre *mode* est un scalaire (chaîne de caractères) comportant un ou deux caractères indiquant le mode d'ouverture :

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

Les habitués du shell retrouveront certaines notations connues.

Par exemple `open(my $fd1,'<','index.html')` ouvre le fichier `index.html` en lecture et `open(my $fd2,'>','index.html')` l'ouvre en écriture-écrasement (c'est-à-dire que le fichier sera vidé avant que le curseur ne soit placé au début du fichier).

Cette fonction `open` renvoie une valeur booléenne vrai ou faux indiquant le bon déroulement ou non de l'opération. Il est très important de tester les valeurs de retour des fonctions manipulant les fichiers (cela est vrai, quel que soit le langage), car on ne peut jamais être sûr de rien. Voici deux exemples de tests de la valeur de retour d'`open` :

```
if( ! open($fd,'>>','data.txt') ) {
   exit(1);
}
```

Dans ce premier exemple, on tente d'ouvrir en ajout un fichier `data.txt` ; en cas d'impossibilité, on appelle la fonction `exit` qui met fin au programme (la valeur 1 signale une erreur ; la valeur 0 signalant la fin normale du programme, les autres valeurs sont utilisées pour signaler au shell appelant une erreur). Vous noterez que l'utilisateur qui exécute le programme n'est pas informé de la cause de l'échec ; le programme se termine, tout au plus sait-il qu'il y a eu un problème avec l'ouverture de ce fichier, mais il n'en connaît pas la cause. L'exemple suivant va nous permettre de lui afficher un joli message d'erreur :

```
open($fd2,'<','/tmp/$a') or die("open: $!");
```

Nous cherchons ici à ouvrir en lecture le fichier dont le nom serait la concaténation de la chaîne `/tmp/` et du contenu de la variable `$a`. La fonction `die` met fin au programme comme `exit` le ferait, mais affiche en plus le paramètre qu'on lui passe. En l'occurrence, le paramètre fourni est la chaîne de caractères comportant le nom de la fonction qui cause l'échec (on aurait pu ajouter le nom du fichier) ainsi que la variable `$!`. En contexte de chaîne de caractères, cette variable magique `$!` contient le message errno 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.

La syntaxe `open() or die();` ainsi que sa signification proviennent de l'évaluation paresseuse du `or`. En effet, dans l'expression (a `or` b) si a est vrai, il n'est pas nécessaire d'évaluer b pour connaître la valeur de l'expression. C'est ce qu'il se passe ici : si `open()` renvoie vrai, il n'est pas nécessaire d'évaluer `die()`.

Les descripteurs de fichier sont d'une espèce étrange en Perl, le débutant s'abstiendra de chercher à trop comprendre les mécanismes sous-jacents. Ce que l'on pourrait dire, c'est qu'il n'est pas nécessaire de déclarer un descripteur de fichier, la fonction `open` valant déclaration.

7.5 Lecture, écriture et fermeture de fichier

Une fois un fichier ouvert, il nous 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.

```
$l = <$fd>;
```

Cette instruction lit la prochaine ligne disponible du fichier FIC. Vous noterez bien que l'opérateur chevrons (*diamond operator* en anglais) est ici en contexte scalaire. En contexte de liste, il renvoie la liste de toutes les lignes restant dans le fichier :

```
@t = <$fd>;
```

Cette instruction « absorbe » toutes les lignes du fichier dans une liste qui est placée dans le tableau `@t`.

Pour itérer sur les lignes d'un fichier, il est courant de faire ainsi :

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

La boucle `while` donne pour valeur à la variable `$l` une à une toutes les lignes du fichier. La fonction `chomp` supprime le dernier caractère s'il s'agit d'un retour à la ligne. La variable spéciale `$.` vaut le numéro de la ligne courante du dernier fichier lu (ici `$fd`).

Si vous utilisez la variable spéciale omniprésente `$_`, la construction

```
while( defined( $_ = <$fd> ) )
```

peut même s'abréger en :

```
while( <$fd> )
```

Pour écrire dans un fichier, nous allons utiliser les fonctions `print` et `printf` que nous avons déjà vues. Elles prennent en premier argument le descripteur de fichier :


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

Il faut noter qu'il n'y a pas de virgule après le descripteur de fichier (il ne faut pas en mettre !). Les parenthèses sont comme toujours optionnelles autour des arguments, mais permettent de lever certaines ambiguïtés. La fonction `printf` fonctionne comme `printf` ou `fprintf` du C et ne sera donc pas détaillée ici (voir `man~3~printf`).

Pour fermer un descripteur de fichier (et donc vider les buffers associés), il faut faire appel à la fonction `close` :

```
close( $fd );
```

À noter que si vous réutilisez un descripteur de fichier dans un `open` sans faire de `close` au préalable, Perl ne rouspétera pas et fermera consciencieusement le premier fichier avant d'ouvrir le deuxième.

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 ;
- `ARGV` : ce descripteur est un peu spécial, mais souvent bien pratique. Les lignes lues sont celles des fichiers de la ligne de commande (donc les arguments passés au programme sont considérés comme des noms de fichier) ; si le programme est lancé sans argument, l'entrée standard est lue. N.B. Vous pouvez écrire soit `<ARGV>` soit `<>` La variable spéciale `$ARGV` contient le nom du fichier en cours de lecture.

Discutons un peu de la manipulation de fichiers binaires. Les exemples de lecture de fichiers donnés jusqu'ici ne conviennent pas à de tels fichiers, mais plutôt à des fichiers contenant du texte. Vous pouvez, pour cela, utiliser la fonction `getc` qui renvoie le prochain caractère disponible : `$c = getc($fd);`. Vous pouvez aussi faire usage de la fonction `read` qui lit un nombre déterminé de caractères : `$tailleLue = read(~$fd,~$tampon,~$tailleÀLire );`. 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. C'est pour cela que l'on récupère la valeur de retour de `read`.

Pour écrire des données non textuelles dans un fichier, vous pouvez tout à fait utiliser les fonctions `print` et `printf`, car les chaînes de caractères de Perl peuvent contenir le caractère de code ASCII zéro. On notera que la fonction `write` existe, mais n'est pas l'inverse de `read`.

S'il vous est nécessaire d'utiliser les fonctions d'entrée/sortie bas niveau, voici leurs noms en Perl : `sysopen`, `sysread`, `syswrite` et `close`.

7.6 Deuxième exemple

Voici le prolongement de l'exemple donné pour les tables de hachage. Nous n'allons plus considérer que les mots sont contenus dans un tableau, mais nous allons les extraire d'un fichier.

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

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

On effectue une boucle `while` sur les lignes du fichier. Chaque ligne est alors découpée en mots par la fonction `split` (`\W+` correspond aux suites de caractères non alphanumériques, nous verrons cela dans la suite lorsque nous étudierons les expressions régulières). Chaque mot est mis en minuscules au moyen de l'opérateur `tr` (que nous expliquerons avec les expressions régulières).

7.7 Exécution de commandes 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* par exemple :

```
open(my $fd1,"ls|")
open(my $fd2,"df -HT $device|")
```

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(my $fd3,"|gzip > $a.gz")
open(my $fd4,"|mail robert\@bidochon.org")
```

Quoi de plus simple ?

Chapitre 8

Expressions régulières

Nous abordons ici un sujet très riche en développements : les expressions régulières. Perl en tire une partie de sa grande puissance pour l'analyse et le traitement des données textuelles. La lecture de cette partie du document peut aussi intéresser toute personne utilisant `grep`, `sed`, Python, PHP, C, C++ et même Java.

Atout important de Perl par rapport à d'autres langages, les expressions régulières permettent de manipuler le texte de façon très puissante et très concise. L'acquisition de leur maîtrise peut s'avérer difficile au début, mais en vaut très largement la chandelle, aussi bien pour programmer en Perl que pour utiliser les outils classiques du shell ou les autres langages précédemment cités.

Au niveau vocabulaire, on utilise en anglais le terme *regular expression* (souvent abrégé en *regexp*, voire *regex*), ce qui a donné en français une traduction correcte "expressions rationnelles" et une traduction mot à mot "expressions régulières". La seconde est entrée dans les mœurs et sera donc utilisée ici.

On retrouve les expressions régulières dans certaines fonctions Perl que vous connaissez déjà, comme `split` ou `grep` ; mais elles existent aussi par le biais d'opérateurs spécifiques.

8.1 Fonctionnalités

Il existe deux types principaux de fonctionnalités dans les expressions régulières : la correspondance (*pattern matching* en anglais : *pattern*=motif, *matching*=correspondance) et la substitution.

La correspondance est le fait de tester (vérifier) si une chaîne de caractères comporte un certain motif. Par exemple, on pourrait se poser les questions suivantes et y répondre par un *match* : la variable `$v` commence-t-elle par un chiffre ? Comporte-t-elle au moins deux lettres majuscules ? Contient-elle une sous-chaîne répétée deux fois d'au moins cinq caractères ? Etc.

Sa syntaxe est la suivante : `m/`*motif*`/`

Le `m` indique que nous voulons faire un *match*, les slashes (`/`) servent à délimiter le motif recherché (on verra plus loin comment utiliser d'autres séparateurs).

Cette fonctionnalité nous permettra aussi d'extraire des sous-chaînes d'une variable donnée sans la modifier. Par exemple, si je veux récupérer le premier nombre que comporte `$v`, j'utiliserai aussi la correspondance.

La substitution permet de faire subir des transformations à la valeur d'une variable. Par exemple : remplacer dans la variable `$v` toutes les sous-chaînes `toto` par `titi`. Supprimer de `$v` tous les mots entre guillemets. Etc.

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

Le `s` indique que nous voulons faire une substitution, les slashes (`/`) servent à délimiter le motif recherché ainsi que la chaîne de remplacement.

8.2 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`. On dit alors que la variable est "liée" à l'expression régulière. Cette expression vaut vrai ou faux ; nous l'utiliserons donc très souvent dans une structure de contrôle de type `if` :

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

Dans les cas où le test est vrai, c'est-à-dire si la variable contient le motif (ici si `$v` contient `sentier`), les instructions seront exécutées.

Par cette opération de bind, nous venons de lier une expression régulière à une variable. Par défaut, une telle expression s'applique à la variable `$_` (comme beaucoup de fonctions Perl).

De la même façon,

```
$v =~ s/voiture/pieds/;
```

remplace la première occurrence de `voiture` dans la variable `$v` par `pieds` (on verra plus loin comment remplacer toutes les occurrences). Le reste de `$v` n'est pas modifié. Le point-virgule indique la fin de l'instruction.

Pour la correspondance, il existe aussi l'opérateur `!~` qui équivaut à `=~` suivi d'une négation de l'expression.

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

est plus concis et est équivalent à

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

Les instructions sont exécutées si `$w` ne contient pas la chaîne `pieds`.

8.3 Caractères

Dans cette sous-partie et dans les suivantes, nous allons voir quels sont les motifs utilisables dans les expressions régulières.

Dans le cas général, un caractère vaut pour lui-même ; comprenez que lorsque l'on utilise l'expression régulière `m/a/` on vérifie si la variable (ici non citée) contient le caractère `a`. Cela semble évident, mais il est bon de le dire.

En effet, pour certains caractères spéciaux, cela n'est pas le cas. Ces caractères ont un rôle particulier dans les expressions régulières (nous allons voir cela dans la suite). Si vous avez besoin

de rechercher ces caractères, il faut donc les déspécifier au moyen d'un anti-slash (\). Voici la liste de ces caractères : `\ | ( ) [ ] { } ^ $ * + ? .`

Il faut ajouter à cette liste le caractère choisi comme séparateur. Pour le moment, seul le slash (`/`) est utilisé dans ce document, mais nous verrons plus tard qu'il est possible d'en changer.

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 ; en voici quelques exemples :

Motif	Caractère
`\n`	saut de ligne
`\r`	retour chariot
`\t`	tabulation
`\f`	saut de page
`\e`	échappement

Seuls les plus utiles sont présentés ici.

8.4 Ensembles

Le caractère `.` (point) correspond à un caractère quel qu'il soit (sauf `\n` (ce comportement peut être changé : nous verrons cela plus loin)). Cela signifie qu'à l'emplacement de ce point dans le motif pourra (devra) correspondre un caractère quelconque dans la variable. Par exemple, le motif `m/t.t./` reconnaîtra toute variable comportant une lettre `t` suivie d'un caractère quelconque, puis une autre lettre `t`, puis un autre caractère quelconque ; par exemple toute variable comportant une des chaînes suivantes correspondra au motif : `tata`, `t%tK`, `tot9`...

Vous comprenez pourquoi il faut déspécifier le caractère point avec un anti-slash si vous voulez chercher un point littéral : sans cela un point *matche* avec n'importe quel caractère.

Le motif `[`*caractères*`]` matche un caractère parmi ceux présents entre crochets. Par exemple `[qwerty]` peut reconnaître une de ces six lettres. Le motif `m/t[oa]t[ie]/` reconnaîtra toute variable comportant une des quatre chaînes suivantes : `toti`, `tati`, `tote` ou `tate`. On comprendra aisément que si un caractère est présent plusieurs fois dans cette liste, cela a le même effet que s'il était présent une seule fois : `[aeiouyie]` est équivalent à `[aeiouy]`.

Il est possible de définir des intervalles de caractères dans ces ensembles. Par exemple `a-z` équivaut aux 26 lettres minuscules de l'alphabet. Par exemple `[2a-zR]` entrera en correspondance avec toute lettre minuscule ou bien avec le 2 ou bien avec le R majuscule. On peut aussi utiliser les ensembles `A-Z` ou `0-9` ; par extension tout intervalle est envisageable, par exemple `R-Z` ou toute autre combinaison tant que le numéro ASCII du premier caractère est inférieur à celui du second. Autre exemple, le motif `[ -~]` correspond à un caractère ASCII imprimable et de numéro inférieur à 127.

Un intervalle peut prendre place au milieu d'un motif quelconque :
`m/tot[a-zA0-9]V/` matche `totaV`, `totbV`... `totzV`, `totAV`, `tot0V`... `tot9V`.

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. Par exemple `[a-z4-]` matche soit une minuscule, soit un 4, soit un tiret.

Le caractère `^` (accent circonflexe) a un rôle particulier s'il est placé en début d'intervalle ; il prend le complémentaire de l'ensemble, il faut le lire "tout caractère sauf...". Par exemple `[^ao]` matche tout caractère sauf le `a` et le `o`. Le motif `[^0-9]` matche tout caractère non numérique.

Nous verrons un peu plus loin qu'il existe des raccourcis pour les ensembles les plus courants.

8.5 Quantificateurs

Les quantificateurs s'appliquent au motif atomique (c'est-à-dire le plus petit possible) le précédant dans l'expression régulière. Ils permettent de spécifier un nombre de fois que 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, avec `a`, `aa`, `aaa`, `aaaa`...

Quand je dis qu'un quantificateur s'applique au motif atomique le plus petit possible, je veux dire par là que dans l'expression régulière `m/za*/` l'étoile s'applique uniquement à la lettre `a` et non au mot `za`. Nous verrons plus loin comment faire cela.

Il est par ailleurs important de noter qu'un tel quantificateur est par défaut gourmand, c'est-à-dire qu'il se met en correspondance avec le plus de caractères possible dans la variable liée. Cela a son importance dans le cas d'une substitution : si la variable `$v` contient la chaîne `vbaaal`, et si on effectue l'instruction suivante : `$v =~ s/ba*/hello/;` la chaîne matchée par la première expression `ba*` sera `baaa` (le quantificateur matche le plus de caractères possible) et la substitution aura pour effet de donner pour valeur `vhellol` à la variable `$v`.

Voici un tableau des quantificateurs :

	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 m et n 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}`.

Dans les exemples précédents, tous les quantificateurs sont appliqués à un caractère. On peut les appliquer à tout motif, par exemple à un ensemble : `m/[0-9-]{4,8}/` recherche une chaîne comportant entre quatre et huit caractères numériques ou tirets contigus.

8.6 Ensembles (suite)

Nous allons ici énumérer un certain nombre de raccourcis pour des ensembles courants :

- `\d` : un chiffre, équivalent à `[0-9]` (d comme digit, chiffre en anglais) ;
- `\D` : un non numérique, équivalent à `[^0-9]`
- `\w` : un alphanumérique, équivalent à `[0-9a-zA-Z_]` (w comme word, c'est un caractère d'un mot) ;
- `\W` : un non-alphanumérique, équivalent à `[^0-9a-zA-Z_]` ;
- `\s` : un espacement, équivalent à `[ \n\t\r\f]` (s comme space) ;
- `\S` : un non-espacement, équivalent à `[^ \n\t\r\f]`.

On remarquera qu'un ensemble et son complémentaire sont notés par la même lettre, l'une est minuscule, l'autre majuscule.

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.

8.7 Regroupement

Si dans notre exemple précédent, nous souhaitons rendre optionnelle la partie décimale, on pourrait écrire : `m/[+-]?\d+\.?\d*/` rendant ainsi non obligatoire la présence du point et celle des chiffres après la virgule. Le problème de cette expression est que la présence du point et de ces chiffres sont décorrélées : l'expression régulière reconnaîtra un nombre où l'une de ces deux parties serait présente et l'autre absente. Or ce que l'on veut, c'est que le point et les chiffres qui le suivent soient rendus solidaires dans l'absence ou la présence.

Pour cela nous allons utiliser des parenthèses pour effectuer un regroupement entre plusieurs motifs (ici le point et les chiffres) pour leur appliquer conjointement le même quantificateur. L'expression régulière
`m/[+-]?\d+(\.\d+)?/` reconnaît donc les nombres tels que nous les souhaitons.

Pour marquer la mémoire de mes étudiants, j'aime à leur dire que
`m/meuh{3}/` permet de meugler longtemps et que `m/(meuh){3}/` de meugler plusieurs fois !

8.8 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`.

De la même façon, l'expression `m/Fred|Paul|Julie Martin/` reconnaît les chaînes comportant soit `Fred`, soit `Paul`, soit `Julie Martin` mais rien n'oblige `Fred` à s'appeler `Fred Martin` ni `Paul` à s'appeler `Paul Martin`, comme on aurait sans doute aimé que cela se fasse (dans ces deux derniers cas, seul le prénom est reconnu, pas le nom). Pour cela, vous l'avez compris, un regroupement est nécessaire. L'expression régulière `m/(Fred|Paul|Julie) Martin/` reconnaît les trois frères et sœur de la famille Martin.

8.9 Assertions

Une assertion marque une position dans l'expression, elle ne correspond à aucun caractère (aucune "consommation" de caractères n'est effectuée).

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 signe `^` a donc plusieurs rôles. S'il est au début d'un ensemble entre crochets, il permet d'en prendre le complémentaire ; s'il est au début de l'expression régulière, il marque le début de la chaîne. On veillera à ne pas les confondre.

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`.

Ces deux assertions sont les plus courantes. Il en existe d'autres dont `\b` qui marque un début ou une fin de mot ; c'est-à-dire entre `\w` et `\W` (ou entre `\w` et une fin ou début de chaîne). Par exemple `m/\btoto\b/` matche "`toto`", "`toto autreMot`", "`unMot toto autreMot`", etc. mais pas "`unMot totoMotCollé`" car le deuxième `\b` ne peut pas être vrai entre la lettre `o` et la lettre `M`.

8.10 Références arrières

Le regroupement au moyen des parenthèses est dit mémorisant. Cela signifie que l'expression matchée par ce regroupement est gardée en mémoire par le moteur d'expressions régulières et qu'elle pourra servir à nouveau dans la suite de l'expression.

L'exemple typique consiste à se demander si une variable contient le même mot répété deux fois. L'expression `m/\w+.*\w+/` ne saurait nous satisfaire ; en effet elle matche toute valeur comportant deux mots pouvant être différents. La solution est d'utiliser les notations `\1`, `\2`, etc. qui font référence aux sous-chaînes matchées par (respectivement) la première, la deuxième, etc. expression entre parenthèses (il n'est pas possible d'accéder à une expression au-delà de `\9`, mais cela nous donne déjà une expression très lourde à gérer).

Par exemple, la réponse à notre problème de deux occurrences d'un même mot est la suivante : `m/(\w+).*\1/` Le `\w+` matchera un mot, les parenthèses mémoriseront la valeur alors trouvée, le `.*` permet comme avant qu'il y ait un nombre indéfini de caractères quelconques entre les deux occurrences, enfin `\1` fait référence à la valeur trouvée par le `\w+` précédent.

Autre exemple basique, `m/(.+), (.+), \2 et \1/` matchera une chaîne de caractères comportant un certain premier motif suivi d'une virgule et d'une espace, puis un certain second motif également suivi d'une virgule et d'une espace, puis ce second motif doit être répété suivi d'une espace, du mot `et` puis d'une autre espace et enfin du premier motif.

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`'.

Vous allez me dire : mais cela signifie que dès que l'on fait un regroupement, le moteur d'expressions régulières mémorise la valeur et si l'on n'utilise pas certains regroupements et que d'autres sont au-delà de 9, on ne peut donc pas s'en servir... Je vais alors vous dire : il existe un regroupement non mémorisant ! La notation `(?:`*motifs*`)` permet de regrouper les motifs (pour leur appliquer le même quantificateur par exemple) sans pour autant qu'une mémorisation n'ait lieu.

Par exemple `m/(.*) (?:et )+(.*) avec \1 \2/` matchera par exemple les valeurs suivantes : "`Paul et Julie avec Paul Julie`" et
"`lala et lili avec lala lili`"

8.11 Variables définies

Ces variables spéciales `$1`, `$2`, etc. sont aussi accessibles *après* l'expression régulière (jusqu'à la fin du bloc courant ou une autre expression régulière). Elles correspondent bien sûr aux sous-chaînes matchées entre parenthèses. Nous pouvons nous en servir pour extraire certaines sous-chaînes et les utiliser ensuite.

Il existe aussi trois autres variables, dont je vous déconseille l'usage, mais qui peuvent être intéressantes :

- `$&` vaut toute la sous-chaîne matchant ;
- `$‘` vaut toute la sous-chaîne qui précède la sous-chaîne matchant ;
- `$’` vaut toute la sous-chaîne qui suit la sous-chaîne matchant.

Je vous déconseille en effet l'usage de ces trois variables spéciales, car leur présence dans un script active pour tout le script des mécanismes particuliers dans le moteur d'expressions régulières, qui ont pour effet secondaire d'en ralentir fortement la vitesse d'exécution. Si vous avez besoin de ces variables dans un petit script qui ne sert qu'à cela, pas de problème pour les utiliser, mais évitez leur usage dans un projet de plusieurs milliers de lignes ou dans un script CGI appelé dix fois par seconde.

Voici un exemple :

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

Il est bon de savoir que cela est possible sans obligatoirement se souvenir du nom de toutes les variables.

8.12 Valeurs de retour de m//

Je vous ai dit jusqu'ici que l'opérateur de correspondance `m//` retournait vrai ou faux ; cela est exact en contexte scalaire. C'est par exemple le cas lorsqu'on écrit :

```
if( $w =~ m/motif/ ) { ... }
```

On parle alors de correspondance.

Mais en contexte de liste, cet opérateur retourne la liste des éléments matchés entre parenthèses (les fameux `$1`, `$2` etc, et cela sans limite à 9). Par exemple :

```
($x,$y) = ( $v =~ m/^(A+).*(B+)$/ );
```

place dans `$x` les caractères `A` du début de la chaîne `$v` et dans `$y` la suite de caractères `B` terminant la chaîne. On parle ici d'extraction.

Il se peut tout à fait que cette opération échoue (en cas d'absence des lettres aux endroits attendus par exemple). Cet usage peut être combiné avec l'utilisation d'un test. On peut en effet écrire :

```
if( ($x,$y) = ( $v =~ m/^(A+).*(B+)$/ ) ) { ... }
```

auquel cas, on n'exécute les instructions du `if` que si `$v` comporte au moins un `A` en son début et un `B` à sa fin. Dans ce cas, les variables `$x` et `$y` reçoivent les valeurs entre parenthèses. On a alors combiné correspondance et extraction.

8.13 Exemples de problèmes

Dans cette partie, je vais vous présenter différents petits exercices pratiques sur les expressions régulières. Les solutions se trouvent un peu plus loin. Essayez de ne pas vous précipiter pour les lire, prenez le temps de chercher dans ce qui précède ce qu'il vous faut pour résoudre les problèmes.

Que font les instructions suivantes ?

- `if( $v =~ m/\w+ \d* ?:/ ) { ... }`
- ```
  if( $v =~ m/^"([a-z]{4,})",/ ) {
     print "$1\n";
  }
  ```
- ```
  if( $v =~ m/([a-z]+)[a-z]*\1/ ) {
     print "$1\n";
  }
  ```
- `($n,$m) = ( $v =~ m/(\w+)=(\d+)/ );`
- ```
  if( ($n,$m) = ( $v =~ m/(\w+)=(\d+)/ ) ) {
     print "$n $m\n";
  }
  ```
- `$v =~ s/^ServerRoot/DocumentRoot/;`
- `$v =~ s/^C="([^"]*)"/D='$1'/;`
- `$v =~ s/ +/ /;`

Écrivez les instructions réalisant les actions suivantes :

- Vérifier que `$v` comporte `velo`.
- Vérifier que `$v` finit par une lettre majuscule.
- Vérifier que `$v` comporte deux fois de suite un même nombre (séparé par un signe d'opération mathématique).
- Extraire de `$v` chacun des deux premiers caractères.
- Extraire de `$v` les deux premiers mots.
- Extraire de `$v` le dernier caractère non numérique.
- Remplacer dans `$v` `rouge` par `bleu`.
- Supprimer de `$v` les espaces en fin de chaîne.
- Supprimer les guillemets autour du nombre entier de `$v`.

Je relève les copies dans 30 minutes `;-)))`

8.14 Solutions des problèmes

Voici les solutions de la première partie des problèmes :

- `if( $v =~ m/\w+ \d* ?:/ ) { ... }`
  On vérifie que `$v` comporte un mot d'une lettre ou plus (`\w+`) suivi d'une espace, puis éventuellement d'un nombre (`\d*`), puis d'une espace optionnelle (`?`) et enfin du signe deux-points. Si c'est le cas, les instructions du `if` sont exécutées.


  ```

- ```
  if( $v =~ m/^"([a-z]{4,})",/ ) {
     print "$1\n";
  }
  ```
  On vérifie que `$v` commence par un guillemet, suivi d'au moins quatre lettres minuscules (que l'on mémorise), d'un autre guillemet puis d'une virgule. Si la variable est du bon format, ces quatre lettres (ou plus) sont affichées.
- ```
  if( $v =~ m/([a-z]+)[a-z]*\1/ ) {
     print "$1\n";
  }
  ```
  On recherche quelque chose de la forme : une suite de caractères en minuscules (au moins 1), puis une deuxième suite de caractères en minuscules (éventuellement aucun) et enfin la même suite de caractères que la première suite. On cherche donc un mot (suite de lettres) dont un certain nombre de lettres se répètent. Si la variable `$v` comporte un tel mot, on affichera ces lettres répétées.
- `($n,$m) = ( $v =~ m/(\w+)=(\d+)/ );`
  On recherche une suite alphanumérique, un signe égal puis un nombre. Il s'agit d'une affectation. La variable et le nombre sont respectivement affectés aux variables `$n` et `$m`.
- ```
  if( ($n,$m) = ( $v =~ m/(\w+)=(\d+)/ ) ) {
     print "$n $m\n";
  }
  ```
  Si la variable `$v` est du format précédemment cité, on affiche la variable et le nombre de l'affectation.
- `$v =~ s/^ServerRoot/DocumentRoot/;`
  On remplace `ServerRoot` par `DocumentRoot` s'il est en début de chaîne.
- `$v =~ s/^C="([^"]*)"/D='$1'/;`
  On recherche en début de chaîne une sous-chaîne `C="`*motif*`"` dont *motif* ne comporte pas de `"`. Tout cela est remplacé par `D='`*motif*`'` où *motif* est inchangé.
- `$v =~ s/ +/ /;`
  Remplace dans `$v` la première suite d'espaces par une seule espace.

Voici les solutions de la seconde partie des problèmes :

- Vérifier que `$v` comporte `velo`.
  Pas trop dur : il s'agit d'un simple match.

  ```
  if( $v =~ m/velo/ ) { ... }
  ```
- Vérifier que `$v` finit par une lettre majuscule.
  Match ici aussi. Le dollar nous permet de nous "accrocher" en fin de chaîne :

  ```
  if( $v =~ m/[A-Z]$/ ) { ... }
  ```
- Vérifier que `$v` comporte deux fois de suite un même nombre (séparées par un signe d'opération mathématique).
  Encore un match. Nous cherchons un nombre `\d+` que nous mémorisons (parenthèses). Un signe doit suivre (il faut déspécifier le signe de la division, car il est aussi le séparateur de l'expression régulière). Finalement le même nombre qu'avant doit être présent (`\1`) :
  ```

```
if( $v =~ m/(\d+)[+*\/-]\1/ ) { ... }
```

- Extraire de `$v` chacun des deux premiers caractères.
  Nous allons utiliser un match pour faire de l'extraction : on se place en début de chaîne avec `^`, on prend un caractère avec `.` que l'on mémorise, puis de la même façon pour le deuxième. Vous noterez que, dans ce cas, l'usage de la fonction `substr` est possible (et indiqué...)

  ```
  ($prem,$deux) = ( $v =~ m/^(.)(.)/ );
  ```

- Extraire de `$v` les deux premiers mots.
  La méthode est la même que pour l'exemple précédent. Le seul point un peu délicat à voir, c'est qu'entre deux mots (`\w+`), il doit forcement y avoir des caractères "non-mot" (`\W+`) :

  ```
  ($prem,$deux) = ( $v =~ m/^\W*(\w+)\W+(\w+)/ );
  ```

- Extraire de `$v` le dernier caractère non numérique.
  De la même façon, après le dernier caractère non numérique, il n'y a que des numériques (zéro ou plus), le dollar pour se placer à la fin de la chaîne :

  ```
  ($c) = ( $v =~ m/(\D)\d*$/ );
  ```

- Remplacer dans `$v` `rouge` par `bleu`.
  Facile (seule la première occurrence est remplacée) :

  ```
  $v =~ s/rouge/bleu/;
  ```

- Supprimer de `$v` les espaces en fin de chaîne.
  On va remplacer toutes ces espaces par rien :

  ```
  $v =~ s/ +$//;
  ```

- Supprimer les guillemets autour du nombre entier de `$v`.
  On va faire une substitution, en mémorisant ce fameux nombre :

  ```
  $v =~ s/"(\d+)"/$1/;
  ```

Les fonctionnalités les plus importantes ont été abordées vous voilà parés pour la suite des opérations. Voici d'autres fonctionnalités plus poussées et donc plus intéressantes. Les quantificateurs non gourmands et surtout les options sont des points importants.

8.15 Choisir son séparateur

Il est tout à fait possible de choisir un autre caractère que le slash (`/`) comme séparateur. Il se peut par exemple que nous ayons à manipuler des URL ; dans ce cas, le caractère slash fait partie des motifs que l'on est susceptible de rechercher ; il est de ce fait fort fastidieux de devoir déspécifier chaque slash utilisé, par exemple dans l'expression suivante (ici une version simplifiée de l'expression régulière qui reconnaît les URL) :

```
if( $v =~ m/http:\/\/\w+\/(\w+\/)*\w+\.html/ )
```


Il serait plus lisible de prendre un autre séparateur, le signe égal par exemple :

```
if( $v =~ m=http://\w+/(\w+/)*\w+\.html= )
```

La plupart des caractères est utilisable comme séparateur.

Si vous utilisez le slash comme séparateur, la lettre `m` n'est pas obligatoire pour faire un match : `$v =~ /velo/` est équivalent à `$v =~ m/velo/`.

Libre à vous de choisir le bon séparateur, sachant que dans la grande majorité des cas le slash est utilisé.

8.16 Options

Après le dernier séparateur des opérateurs de correspondance (`m` ou rien) ou de substitution (`s`) 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. Voici la liste de quelques options parmi les plus utiles.

- L'option `i` rend le motif insensible à la casse (minuscules/majuscules) : 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 défaut, l'opérateur `s///` effectue la transformation de la première occurrence du motif recherché et ne va pas plus loin. Si cette option est spécifiée, le moteur d'expressions régulières avancera dans la variable tant qu'il pourra y faire des substitutions.
  Par exemple, l'expression `$v =~ s/ +/ /g;` remplace chaque groupe de plusieurs espaces par une seule (contrairement à un des exercices précédents où l'expression régulière ne remplaçait que la première occurrence du motif trouvé).
  Voyez par exemple le code suivant :
  ```
  $t = $s = "sd et sd";
  $t =~ s/sd/toto/;   # => "toto et sd"
  $s =~ s/sd/toto/g;  # => "toto et toto"
  ```
  L'option `g` est aussi utilisable en correspondance. Elle permet à cet opérateur de fonctionner avec état, c'est-à-dire de poursuivre sa recherche en partant du dernier motif trouvé. On l'utilise typiquement dans une boucle ; voyez cet exemple :
  ```
  my $v = "aatobbtbvvtczz";
  while( $v =~ m/t./g ) {
     print "$&\n";
  }
  ```
  L'affichage effectué est le suivant :
  ```
  to
  tb
  tc
  ```
- 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 `$s` par la valeur de retour de la fonction appliquée à ce nombre.
Autre exemple, avec des options combinées celui-là :
`$s =~ s/0x([0-9a-f]+)/hex($1)/gei;`
transforme tous les nombres hexadécimaux en nombres décimaux dans la variable `$s`.

- L'option `o` a pour effet qu'une seule compilation de l'expression régulière a lieu. En temps normal, à chaque fois que l'interpréteur Perl passe sur une expression régulière, il la compile (pour ceux qui connaissent, il construit l'automate) ; avec cette option, la compilation a lieu une seule fois lors de la première exécution. Le principal avantage est un temps d'exécution plus court pour le programme, si cette expression est utilisée plusieurs fois. Les inconvénients sont une place mémoire occupée (inutilement si l'expression régulière ne sert que peu de fois) et que, si le motif peut changer (voir la suite concernant les variables dans les motifs), ce changement ne sera pas pris en compte.

Il existe deux options (exclusives l'une de l'autre) qui permettent de changer certains comportements sur les débuts et fins de chaînes. Pour les exemples qui suivent, je pose `$s = "mot\nlu";` :

- Par défaut : mode intermédiaire.
  - Les caractères `^` `$` se positionnent en début/fin de chaîne. Par exemple `($s=~m/mot$/)` est faux.
  - Le caractère `.` ne matche pas `\n`. Par exemple `($s=~m/t.lu$/)` est faux.
- Avec l'option `s` : on travaille en ligne unique.
  - Les caractères `^` `$` se positionnent en début/fin de chaîne. Par exemple `($s=~m/mot$/s)` est faux.
  - Le caractère `.` peut matcher `\n`. Par exemple `($s=~m/t.lu$/s)` est vrai.
- Avec l'option `m` : on travaille en ligne multiple.
  - Les caractères `^` `$` se positionnent en début/fin de ligne. Par exemple `($s=~m/mot$/m)` est vrai.
  - Le caractère `.` ne matche pas `\n`. Par exemple `($s=~m/t.lu$/m)` est faux.

8.17 Quantificateurs non gourmands

Posons-nous le problème suivant. Nous avons une chaîne de la forme `"s 'r' g 'e' y"` de laquelle nous souhaitons extraire les chaînes qui sont entre guillemets. La première idée est d'écrire quelque chose comme : `/'.*'/` ce qui n'est pas satisfaisant, car dans notre exemple la chaîne `'r' g 'e'` serait matchée. En effet, je vous avais dit que les quantificateurs consomment le plus de caractères possible, nous voici dans une illustration du phénomène. On parle de quantificateurs gourmands, gloutons, avides ou *greedy* en anglais.

Il existe des quantificateurs dont le comportement est, au contraire, de consommer le moins de caractères possible. On parle alors de quantificateurs non gourmands, économes ou frugaux. Leur notation est la même que celle des quantificateurs que vous connaissez, mais suivie d'un point d'interrogation :

Gourmand	Non gourmand
`*`	`*?`
`+`	`+?`
`?`	`??`
`{n,m}`	`{n,m}?`

Pour revenir à notre exemple, on peut écrire `/'.*?'/` et la correspondance sera effectuée telle que nous la souhaitions.

Vous allez me dire, car vous avez tout compris aux expressions régulières `;-)` , qu'il est possible de faire cela sans utiliser ces quantificateurs non gourmands. Dans notre exemple, on peut tout à fait écrire : `/'[^']*'/` ce qui permet de ne pas accepter de guillemets entre les guillemets. Je répondrai que je suis d'accord avec vous.

Mais voici un autre exemple où les quantificateurs non gourmands nous sauvent la mise. Si cette fois la chaîne a pour valeur
`"s STARTrSTOP g STARTe fSz zSTOP y"` et que nous souhaitons extraire les sous-chaînes placées entre les marqueurs `START` et `STOP`, il nous est fort aisé d'écrire : `/START.*?STOP/`

8.18 Substitution de variables dans les motifs

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 le mot `velo` en fin de chaîne ; le premier dollar concerne la variable `$s`, le second marque la fin de chaîne. Perl sait automatiquement si un `$` correspond à une variable ou à la spécification "fin de chaîne". Il est ainsi possible de rechercher la valeur d'une variable dans une autre. La même chose est possible avec la substitution, aussi bien pour le premier membre que pour le second.

Notez cependant que si la variable substituée contient des caractères spéciaux au sens des expressions régulières, ils seront vus comme tels. Si dans notre exemple, la variable `$s` avait pour valeur la chaîne "`ve(lo`", le moteur d'expression régulière nous signalerait une erreur due à la parenthèse ouvrante qui n'est pas refermée, exactement comme si nous avions écrit :

```
if( $v =~ m/ve(lo$/ ) { ... }  # incorrect
```

Cela peut aussi poser des problèmes de sécurité si le programmeur ne sait pas ce que peut contenir la variable substituée (par exemple si sa valeur provient de l'utilisateur). Il existe pour cela une fonction `quotemeta` qui prend en paramètre une chaîne de caractères et renvoie cette même chaîne en ayant déspécifié les caractères spéciaux.

```
$s = "fds(ds";
$s2 = quotemeta($s);
print "$s2\n";   # affiche  fds\(ds
if( $v =~ m/$s2/ ) { ... }
```

Pensez à toujours utiliser cette fonction lorsque vous voulez placer une variable dans un motif ; cela résout bien des problèmes.

8.19 Opérateur `tr`

Cet opérateur ne concerne pas vraiment les expressions régulières, mais il en est proche. Mettez de côté ce que vous venez d'apprendre sur celles-ci pour lire la suite.

`tr` est un opérateur de translation lettre à lettre (on parle de translittération). Voici sa syntaxe : `tr/`*chaîne1*`/`*chaîne2*`/` ou encore
`y/`*chaîne1*`/`*chaîne2*`/`

Les deux chaînes doivent être de la même longueur, car cet opérateur va remplacer la première lettre de la première chaîne par la première lettre de la seconde chaîne, la deuxième lettre de la première chaîne par la deuxième lettre de la seconde chaîne, etc. Cette transformation a lieu sur la variable liée ou sur `$_` par défaut.

Voici un petit exemple :

```
$s = "azerty";
$s =~ tr/abcde/01234/;
print "$s\n";  # affiche 0z4rty
```

Dans la variable `$s`, tous les a seront transformés en 0, tous les b en 1, etc, tous les e en 4, etc.

Il est possible d'utiliser des intervalles : `$s =~ tr/a-z/A-Z/;` met par exemple le contenu de la variable en majuscules. C'est l'un des seuls usages courants de l'opérateur `tr`.

8.20 Un dernier mot sur les expressions régulières

Les expressions régulières sont un outil très puissant. Les maîtriser ouvre des portes au programmeur. Certes, il est souvent difficile de rentrer dans le jeu, mais cet effort est récompensé par de nouvelles possibilités inimaginables avant.

Les expressions régulières de Perl sont si puissantes et bien pensées que de nombreux langages les implémentent, en se vantant d'être perl5-regexes compliant ! On peut, par exemple, citer la bibliothèque *pcre* du langage C, dont le nom provient des initiales de « Perl-compatible regular expressions »...

Sachez aussi que toutes les fonctionnalités des expressions régulières n'ont pas été traitées ici. Les plus courantes ou importantes le sont, mais il en existe d'autres encore... Sachez de plus que les expressions régulières, c'est bien, mais il faut savoir quand les utiliser et quand ne pas les utiliser.

Chapitre 9

Références

Les références permettent de bâtir des structures complexes et composées : tableau de tableaux ou de tables de hachage et inversement, table de hachage de tableaux ou de tables de hachage...

Le terme de référence en Perl correspond à peu près à celui de pointeur en C et C++ et à celui de référence en Java. Les habitués du C ou C++ noteront que les calculs sur références sont interdits en Perl, ce qui permet d'éviter toutes sortes de problèmes dus à des accès mémoire erronés (plus de `Segmentation fault`).

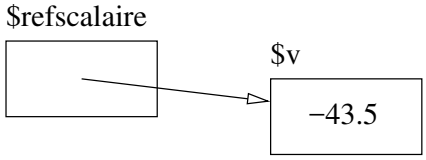
Chaque variable, qu'elle soit scalaire, tableau ou table de hachage, est présente à une position donnée dans la mémoire. Une référence vers une variable est (schématiquement) l'adresse mémoire de cette variable. Une telle référence peut elle-même être stockée dans une variable scalaire.

9.1 Références sur scalaire

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

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

Ici la variable `$refv` (on aurait pu choisir un tout autre nom pour cette variable) est une référence vers la variable `$v`. Une variable de type référence, quel que soit celui de la variable qu'elle référence (scalaire, tableau ou table de hachage), est un scalaire. On peut représenter la relation entre ces deux variables de la façon suivante :



La variable `$refv` contient l'adresse de la variable `$v` (ainsi que l'information consistant à savoir que `$v` est une variable scalaire). On dit que `$refv` *pointe vers* `$v`, car on va pouvoir manipuler `$v` (l'afficher, la modifier, etc.) en utilisant `$refv` ; on représente ce lien par une flèche de `$refv` vers `$v`.

Il nous est alors possible de manipuler `$v` au travers de `$refv`. La notation `$$refv` (donc avec deux dollars) est équivalente à `$v` tant que `$refv` pointe vers `$v`. Autrement dit, la notation `$$refv` équivaut à la variable scalaire pointée par la référence `$refv`. Dit de manière plus prosaïque, on va accéder à la variable qu'il y a "au bout" de la référence (au bout de la flèche du schéma). On dit alors que l'on *déréférence* la variable `$refv`. Pour faire un parallèle avec le langage C, il s'agit de l'équivalent de l'étoile (`*`) appliquée à un pointeur.

Revenons sur notre exemple. Nous déclarons une variable scalaire `$v` que nous initialisons. Nous déclarons ensuite une autre variable scalaire `$refv` à laquelle on affecte l'adresse de `$v` ; `$refv` est donc une référence sur `$v` :

```
my $v = -43.5;
my $refv = \$v;
```

On affiche la valeur de la référence :

```
print "$refv\n";   # affiche SCALAR(0x80ff4f0)
```

On voit bien alors que la référence pointe vers une variable de type scalaire (`SCALAR`) dont l'adresse mémoire est affichée en hexadécimal. Affichons maintenant la variable pointée par `$refv` (c'est-à-dire `$v` ici) :

```
print "$$refv\n";  # affiche -43.5
```

L'affichage effectué est `-43.5` (c'est-à-dire la valeur de `$v`). Cette notation `$$refv` est équivalente à `$v` puisque `$refv` est une référence sur `$v`. Cette équivalence vaut aussi bien lorsque l'on a besoin de la valeur de `$v` (affichage, etc.) que lorsque l'on veut affecter une nouvelle valeur à `$v` :

```
$$refv = 56;       # affecte 56 à $v
```

On affecte 56 à `$$refv`, c'est-à-dire à `$v`. Si on affiche cette variable, on voit bien qu'elle contient cette nouvelle valeur :

```
print "$$refv\n";  # affiche 56
print "$v\n";      # affiche 56
```

Rien de bien sorcier.

9.2 Utilisation des références sur scalaire

À quoi peut bien servir une référence sur un scalaire ? Par exemple à le modifier dans une fonction :

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


Cette fonction prend en argument une référence et affecte 0 à la variable scalaire pointée par cette référence. On pourrait l'utiliser ainsi :

```
f( $refv );
```

Ce qui aurait pour effet de mettre la variable `$v` à la valeur 0. On pourrait aussi écrire directement :

```
f( \$v );
```

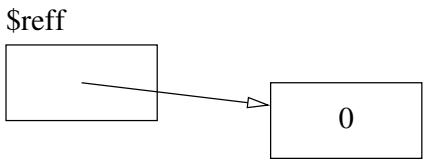
Voici un autre exemple simple d'utilisation des références :

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

Cette fonction `f2` déclare une variable locale `$w` et renvoie une référence vers cette variable. Contrairement à ce qu'il se passe en C, ceci est tout à fait légal et sans risque en Perl. Voici comment utiliser cette fonction :

```
my $reff = f2();
```

La variable scalaire `$reff` devient donc une référence vers une variable scalaire valant 43. Cette variable scalaire valant 43 est l'ancienne variable `$w` de la fonction `f2`. La variable `$reff` pointe donc vers une variable qui n'a plus de nom : dans `f2` elle s'appelait `$w`, mais en dehors de l'appel à cette fonction qui l'a créée, elle n'a plus de nom.



En temps normal, une variable locale à une fonction est détruite lorsque l'on sort de la fonction. Mais tant qu'il existe une référence vers la variable, elle est conservée en mémoire. C'est le garbage collector (ramasse-miette ou glaneur de cellules) qui la libérera lorsque plus aucune référence sur la variable n'existera.

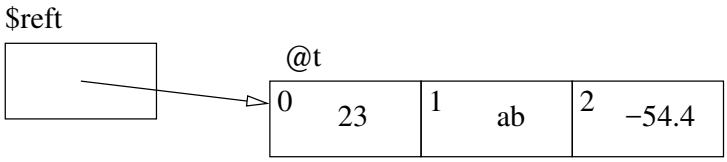
Il faut noter que lors d'un prochain appel à la fonction `f2`, une autre variable `$w` sera créée indépendante de la première ; il n'y aura donc pas d'effet de bord sur la première référence renvoyée. Nous sommes ici en présence d'une fonction qui peut faire office de générateur de références sur scalaire `;-)`

9.3 Références sur tableau

Il est possible de créer une référence sur un tableau. L'opérateur qui permet cela est le même que pour les scalaires ; il s'agit de l'anti-slash (\) appliqué à une variable de type tableau :

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

La variable scalaire `$reft` est donc une référence vers le tableau `@t` :



Pour déréférencer une telle référence, il convient d'utiliser une arobase (`@`) : `@$reft` est équivalent à `@t`. On peut ainsi utiliser la valeur de `@t` en utilisant `$reft` :

```
my @t2 = @$reft;
foreach my $e (@$reft) { .... }
```

On peut aussi modifier `@t` de la sorte :

```
@$reft = (654.7, -9, "bonjour");
```

Si, pour accéder au ième élément de `@t`, il était possible d'écrire `$t[i]`, il est maintenant possible d'écrire `$$reft[i]`. On peut alors dire, de manière schématique et pour fixer les choses, que la notation `$reft` est équivalente au nom `t` de la variable dans toutes les syntaxes utilisant ce tableau (partout où l'on peut écrire `t`, on peut écrire `$reft` à la place, tant que `$reft` pointe sur `@t`). Voici, en effet, un récapitulatif des équivalences de notations :

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

Cette dernière notation `$reft->[i]` est équivalente à `$$reft[i]` et correspond donc au ième élément du tableau référencé par `$reft`. C'est la notation la plus souvent utilisée pour cela ; elle rappelle la même notation flèche (`->`) du langage C.

L'expression

```
$reft->[1] = "coucou";
```

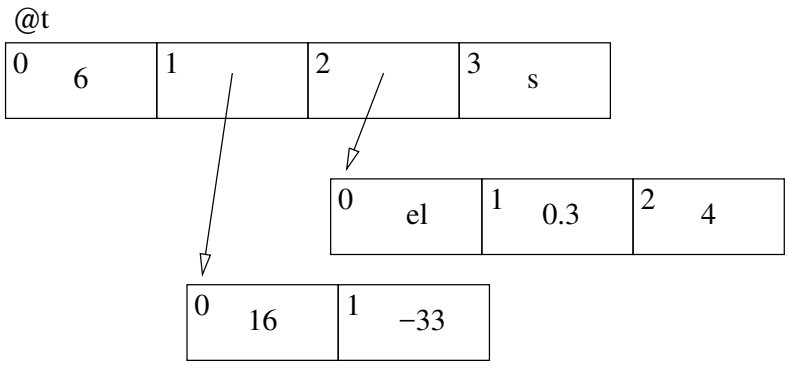
affecte donc à l'élément d'indice 1 du tableau pointé par `$reft` une nouvelle valeur.

Muni des références, il va maintenant nous être possible de créer des tableaux de tableaux. Cela était pour le moment impossible en raison de l'aplatissement des listes. 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" );
```

Le tableau `@t` comporte donc un premier élément valant 6, un deuxième étant une référence vers un tableau à deux éléments, un troisième une référence vers un tableau à trois éléments et enfin un élément valant la chaîne `"s"`.

Ce qui peut se représenter sous la forme suivante :



Vous noterez bien que la syntaxe suivante correspond à la création d'un tableau à sept éléments (aplatissement des listes) :

```
my @t2 = ( 6,
           ( 16, -33 ),
           ( "el", 0.3, 4 ),
           "s" );
```

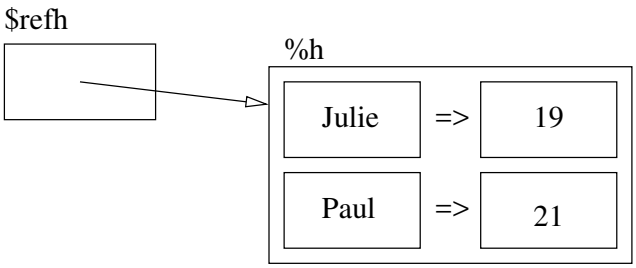
Nous verrons dans la suite d'autres syntaxes pour construire des tableaux de tableaux.

9.4 Références sur table de hachage

De la même façon que pour les scalaires et pour les tableaux, la création d'une référence vers une table de hachage utilise l'opérateur anti-slash (\) :

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

La variable scalaire `$refh` est donc une référence vers la table de hachage `%h` :



Pour déréférencer une référence vers une table de hachage, il faut utiliser le caractère pourcentage (`%`) ; `%$refh` est équivalent à `%h` :

```
my %h2 = %$refh;
foreach my $k (keys %$refh) { ... }
```

Les deux notations suivantes permettent d'accéder à la valeur associée à la clef `Paul` : `$$refh{Paul}` et `$refh->{Paul}` sachant que la seconde est la plus utilisée pour des raisons identiques aux cas des tableaux.

```
$refh->{Jacques} = 33;
```

Voici un récapitulatif des équivalences :

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

Voici une façon d'afficher tous les couples clef/valeur de la table de hachage référencée par `$refh` :

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

Cette notation flèche rend l'expression plutôt lisible.

9.5 Réflexions à propos des références

On a vu que pour déréférencer une référence vers un scalaire, on utilise le caractère dollar (`$`), vers un tableau le caractère arobase (`@`) et vers une table de hachage le caractère pourcentage (`%`). Mais que se passerait-il s'il nous venait à l'idée de ne pas choisir le bon caractère de déréférencement, par exemple d'utiliser le dollar pour une référence sur tableau ou bien le pourcentage pour une référence sur scalaire ? N'arrivions-nous pas à des incohérences comme en C ?

La réponse est non, car l'interpréteur veille au grain. Il refusera de considérer comme un tableau une variable scalaire par exemple. En cas d'incompatibilité de type, un de ces trois messages sera affiché lors de l'exécution et le programme prendra fin :

```
Not a SCALAR reference at script.pl line 23.
Not an ARRAY reference at script.pl line 23.
Not a HASH reference at script.pl line 23.
```

Comme une référence peut pointer vers n'importe quel type de structure (scalaire, tableau, table de hachage), cette vérification ne peut avoir lieu qu'au moment de l'exécution.

De plus, les habitués du langage C seront invités à se mettre une bonne fois pour toutes dans la tête `:-)` qu'il n'y a pas d'arithmétique possible sur les références en Perl. Ajouter 1 à une référence ne correspond pas à pointer vers l'élément d'indice 1 d'un tableau, mais à faire perdre le caractère référence à la variable (elle devient un scalaire comme un autre) :

```
my $v = 45;
my $r = \$v;
print "$r\n"; # affiche SCALAR(0x80fd4c4)
$r++;
print "$r\n"; # affiche 135255237
```

On voit bien ici que la variable `$r` perd son caractère spécifique de référence ; elle a toujours pour valeur l'adresse de la variable (ici incrémentée de 1 : 80fd4c4 est la représentation hexadécimale du nombre 135 255 236), mais n'est plus une référence, il sera donc impossible de la déréférencer.

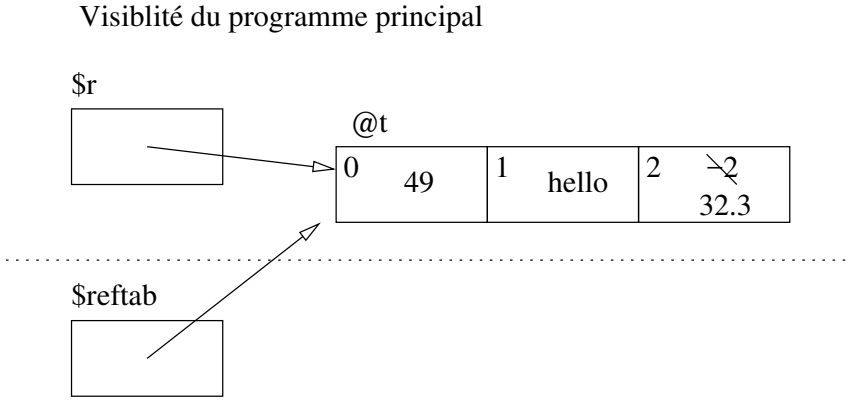
Notez aussi qu'une référence peut changer de valeur, c'est-à-dire de variable pointée et donc de type de variable pointée :

```
my $v = 45;
my @t = ("ee",-2);
my $r = \$v;
$$r = -32;    # modification de $v
$r = \@t;
$r->[0] = 28; # modification de $t[0]
```

Dernière chose importante, si vous passez une référence à une fonction, vous devez bien voir que la copie de la seule référence est effectuée, la structure pointée par la référence ne l'est pas ; vous pourrez donc la modifier dans la fonction :

```
sub f3 {
   my ($reftab) = @_;
   $reftab->[2] = 32.3;
}
my @t = ( 49, "hello", -2 );
my $r = \@t;
f3( $r );
f3( \@t ); # équivalent
# @t est modifié
```

On peut schématiser ce qu'il se passe de la manière suivante :



Visibilité de la fonction f3

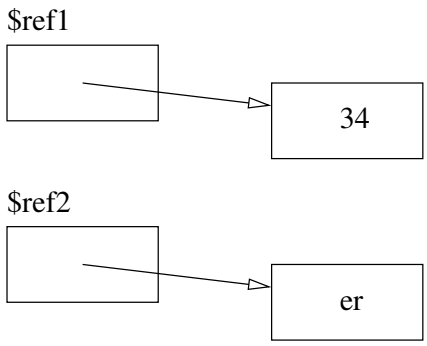
Les deux références pointent donc vers la même zone mémoire.

9.6 Références anonymes vers scalaire

Une référence anonyme est une référence vers une variable qui n'a pas de nom. Nous avons déjà vu comment faire cela pour un scalaire avec une fonction (fonction `f2` un peu avant), mais cela n'est pas la façon de faire la plus simple.

La syntaxe pour créer directement une référence anonyme vers un scalaire est la suivante : *valeur-constante*

```
my $ref1 = \34;
my $ref2 = \"er";
print "$$ref1 $$ref2\n";
```



Une telle valeur est une constante, il est impossible de modifier la valeur pointée par la référence (différence avec le mécanisme de la fonction `f2`) :

```
$$ref1 = "hello";
# Modification of a read-only value attempted at script.pl line 24.
```

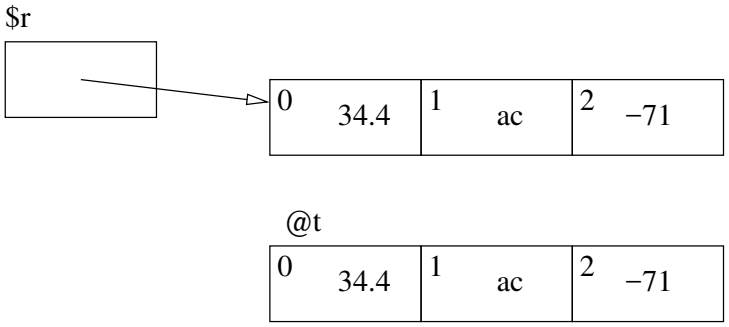
Ce n'est pas dans le cas des scalaires que les références anonymes sont les plus utilisées ; elles le sont bien plus avec les tableaux et les tables de hachage.

9.7 Références anonymes vers tableau

Pour créer une référence anonyme vers un tableau, il faut utiliser la notation suivante : `[`*élément1*`,` *élément2*`,` *élément3*`,` *etc.*`]` est une référence vers un tableau comportant les éléments en question.

```
my $r = [ 34.4, "ac", -71 ];
my @t = ( 34.4, "ac", -71 );
```

La variable `$r` est une référence vers un tableau comportant trois éléments alors que la variable `@t` est un tableau à trois éléments (cette dernière notation nous est déjà familière) :



On accède aux éléments de cette référence anonyme comme pour une référence normale :

```
print "$r->[0]\n";
$r->[2] = 901;
foreach my $e (@$r) { ... }
```

Attention à ne pas écrire `\(2,"er",$v)` si vous voulez créer une référence vers un tableau, car cette syntaxe fait toute autre chose : elle est en fait équivalente à `(\2,\"er",\$v)`, donc à une liste de références, ce qui est fort différent.

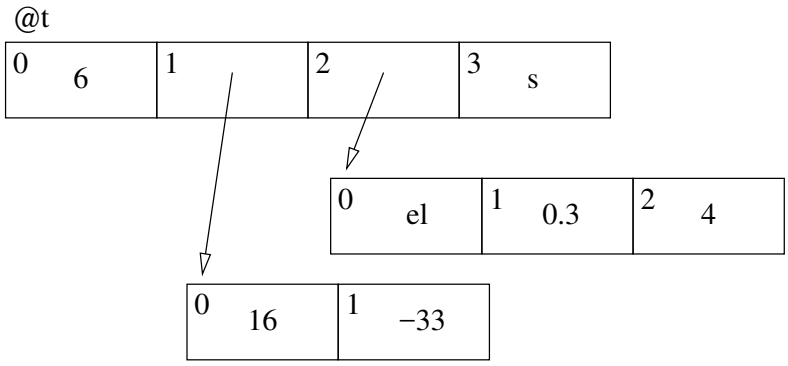
La structure précédemment décrite par

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

peut donc s'écrire de la manière suivante :

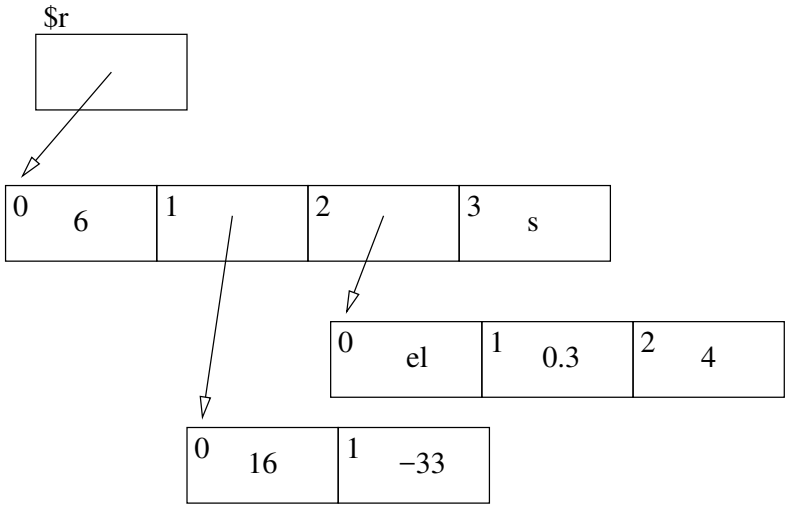
```
my @t = ( 6,
          [ 16, -33 ],
          [ "el", 0.3, 4 ],
          "s" );
```

Ce qui peut se représenter toujours sous la forme suivante :



On peut pousser le vice jusqu'à utiliser une référence pour le premier tableau :

```
my $r = [ 6,
          [ 16, -33 ],
          [ "el", 0.3, 4 ],
          "s" ];
```



Comment accéder aux éléments des différentes profondeurs d'une telle construction ? Il suffit de 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
# $r->[2] est une référence vers tableau
print "$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
```

Ce n'est pas si complexe qu'il n'y paraît pour peu que nous ayons un bon schéma sous la main... Vous noterez que nous faisons usage de l'opérateur flèche (`->`) plutôt que de la syntaxe double-dollar (`$$`).

De plus, si `$r->[1]` est une référence vers tableau, `@{$r->[1]}` est le tableau en question. On peut donc écrire :

```
foreach my $e ( @{$r->[1]} ) { ... }  # Parcourt 16 et -33
```

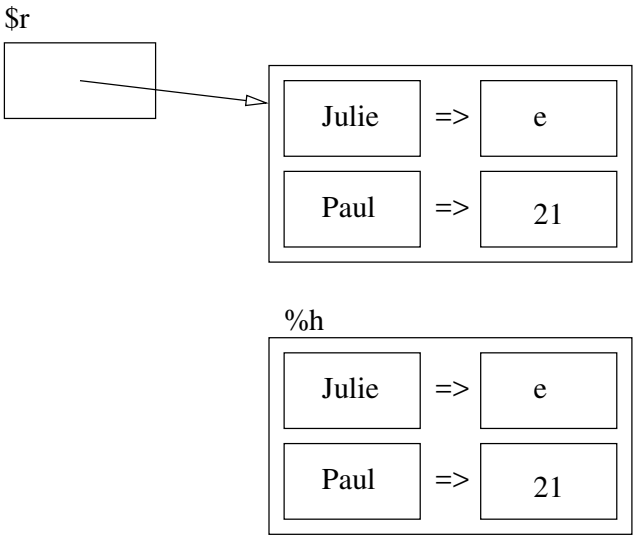
Il faut noter qu'en cas de déréférencements successifs, seule la première flèche est nécessaire : `$r->[2][1]` est équivalent à `$r->[2]->[1]`.

9.8 Références anonymes vers table de hachage

De la même façon, il est possible de créer une référence anonyme vers une table de hachage. La notation `{`*clef1*`=>`*valeur1*`,` *clef2*`=>`*valeur2*`,` *etc.*`}` est une référence vers une table de hachage comportant les couples clef/valeur en question.


```
my $r = { 'Paul' => 21,
          'Julie' => "e" };
my %h = ( 'Paul' => 21,
          'Julie' => "e" );
```

La variable `$r` est une référence vers une table de hachage alors que la variable `%h` est une table de hachage (notation familière) :

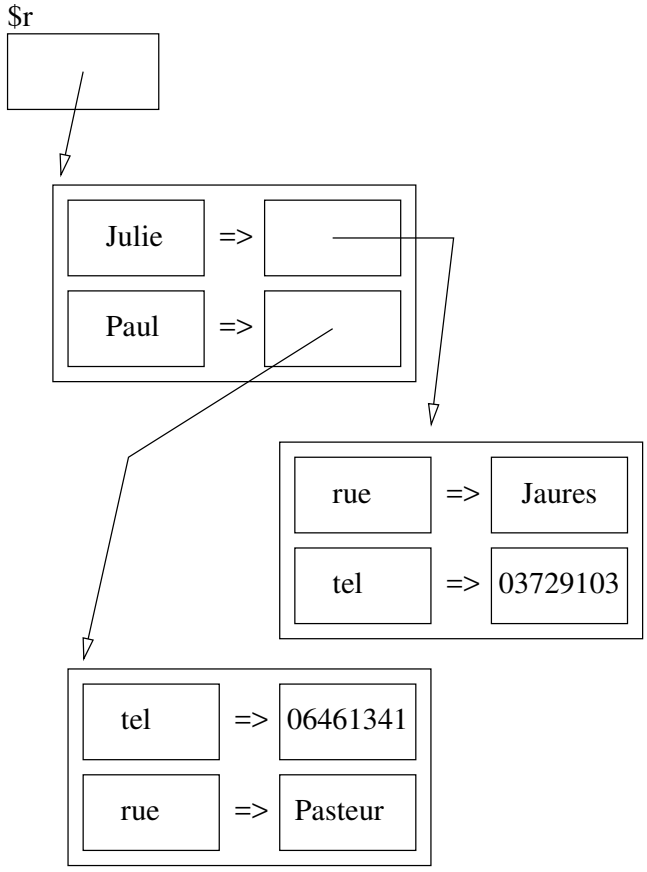


De la même façon qu'avec des tableaux, nous allons mettre sur pied des tables de hachage de tables de hachage :

```
my %h1 = ( 'rue' => 'Pasteur',
           'tel' => '06461341' );
my %h2 = ( 'rue' => 'Jaures',
           'tel' => '03729103' );
my $r = { 'Paul' => \%h1,
          'Julie' => \%h2 };
```

Ou plus directement :

```
my $r = {
   'Paul' =>
       { 'rue' => 'Pasteur',
         'tel' => '06461341' },
   'Julie' =>
       { 'rue' => 'Jaures',
         'tel' => '03729103' }
};
```



Voici comment accéder à tous les éléments d'une telle construction :

```
# $r->{Paul} est une référence vers une table de hachage
print "$r->{Paul}->{tel}\n"; # affiche 06461341
print "$r->{Paul}{tel}\n";   # équivalent
print "$r->{Paul}{rue}\n";   # affiche Pasteur
# $r->{Julie} est une référence vers une table de hachage
print "$r->{Julie}{tel}\n";  # affiche 03729103
print "$r->{Julie}{rue}\n";  # affiche Jaures
```

Il suffit de suivre sur le schéma, d'où l'importance de faire un bon schéma.

9.9 Références anonymes diverses

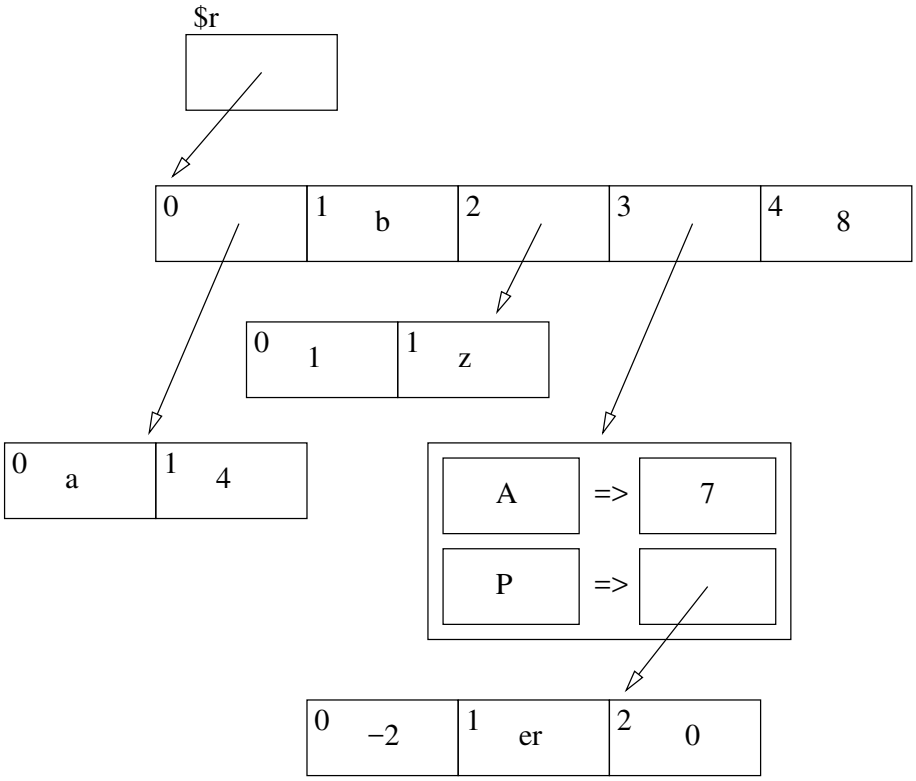
Il est bien sûr tout à fait possible de mélanger les références vers les tableaux et vers les tables de hachage :

```
my $r = [
   ['a',4],
   'b',
   [1,'z'],
   {'P'=>[-2,"er",0],'A'=>7},
```



```
   8
];
```

Voici le schéma correspondant à cette structure (essayer de vous en convaincre !) :



Voici comment accéder à un élément d'une telle structure :

```
print "$r->[3]->{P}->[1]\n"; # affiche "er"
print "$r->[3]{P}[1]\n";     # équivalent
```

Les crochets correspondent à une prise d'indice d'un tableau ; les accolades à la clef d'une table de hachage.

Je peux parcourir le premier tableau du deuxième niveau (celui qui comporte `a` et `4`) de la manière suivante :

```
my $reft = $r->[0];
foreach my $v (@$reft) {
   print "$v\n";
}
```

Je crée une variable `$reft` qui est une référence vers ce tableau, je peux ensuite parcourir `@$reft` qui représente le tableau en question.

Il est possible d'écrire cela sans créer de variable temporaire, la syntaxe est la suivante : `@{`*référence*`}` Ces accolades n'ont rien à voir avec les tables de hachage elles permettent juste de délimiter la référence à laquelle on applique l'arobase. Voici ce que cela donne :

```
foreach my $v (@{$r->[0]}) {
   print "$v\n";
}
```

On fait de la même façon pour une table de hachage, les accolades délimitent la référence à laquelle on applique le pourcentage :

```
foreach my $k (keys %{$r->[3]}) {
   print "$k $r->[3]{$k}\n";
}
```

Commencez-vous à voir l'intérêt de faire des schémas ?

9.10 L'opérateur `ref`

La question qui pourrait maintenant venir à l'esprit est la suivante : comment pourrait-on écrire une boucle sur les éléments du tableau référencé par `$r` et surtout comment savoir de quel type ils sont pour pouvoir les utiliser ? Pour répondre à cette question, voici un nouvel opérateur : `ref()`. Il permet de connaître le type d'une référence.

Cette fonction 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 (c'est un scalaire classique).

On peut alors écrire le code suivant :

```
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";
   }
}
```

L'affichage suivant est effectué :

```
( a 4 )
b
( 1 z )
```



```
P : ARRAY(0x8100c20)
A : 7
8
```

Dans cet exemple, un seul premier niveau de références est exploré. Pour aller au-delà, c'est-à-dire, afficher le tableau associé à la clef `P`, il faudrait concevoir un ensemble de fonctions s'appelant les unes les autres en fonction du type des références rencontrées. Ne vous fatiguez pas à les écrire, il existe déjà une telle fonctionnalité en Perl :

```
use Data::Dumper;
print Dumper($r);
```

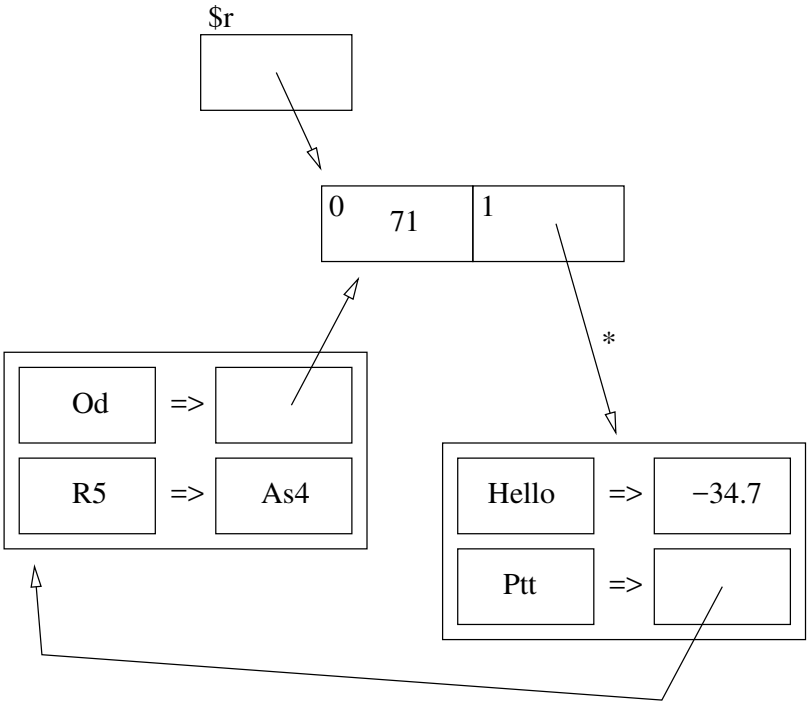
La première ligne ajoute des fonctions à Perl (c'est un peu le `#include` du langage C) et ne doit donc être présente qu'une seule fois dans le programme (plutôt au début). La seconde consiste en l'affichage de la valeur de retour de la fonction `Dumper` : cette fonction renvoie une (longue) chaîne de caractères représentant toute la structure référencée par `$r` :

```
$VAR1 = [
          [
            'a',
            4
          ],
          'b',
          [
            1,
            'z'
          ],
          {
            'P' => [
                      -2,
                      'er',
                      0
                    ],
            'A' => 7
          },
          8
        ];
```

Vous noterez que l'affichage effectué est directement intégrable dans un code Perl.

9.11 Références circulaires

La notion de référence circulaire n'a rien de compliqué ; c'est juste le fait que plusieurs tableaux et/ou tables de hachage et/ou scalaires peuvent se référencer entre eux. Par exemple :



On voit bien alors qu'un cycle de références existe entre ces références. Voici comment écrire cela en Perl :

```
my $r = [
    71,
    {
      "Hello" => -34.7,
      "Ptt" => { "R5" => "As4" }
    }
  ];
$r->[1]{Ptt}{Od} = $r;
```

On comprend bien qu'il n'est pas possible de créer une telle structure en une seule instruction. Comment se comporte `Data::Dumper` dans une telle situation ? Ne va-t-il pas boucler à l'infini ? Eh bien non : il se comporte bien :

```
print Dumper($r);
$VAR1 = [
    71,
    {
      'Ptt' => {
           'R5' => 'As4',
           'Od' => $VAR1
              },
      'Hello' => '-34.7'
    }
  ];
```

`Data::Dumper` se rend compte qu'il passe sur une référence qu'il a déjà parcourue et l'affiche comme telle (`$VAR1`).

Pourquoi vous parler de références circulaires ? Premièrement parce qu'il faut savoir qu'il est possible d'en faire (cela peut être utile de créer des listes chaînées circulaires, etc.). Mais surtout parce qu'elles posent des problèmes au garbage collector dans sa tâche de libération de la mémoire. À la suite de l'exemple précédent, je pourrais écrire :

```
$r = undef;
```

En temps normal, tout ce que `$r` référençait serait libéré. Mais ici, ce n'est pas le cas. En effet, chacun des tableaux et des tables de hachage a encore au moins une référence vers eux, le garbage collector ne se rend pas compte, qu'en fait, les trois objets peuvent être libérés. Nous sommes donc en présence de zones mémoires inaccessibles (aucune variable ne nous permet d'y accéder) et non libérables : cette mémoire est perdue pour le programme ! Elle sera bien sûr libérée quand le programme prendra fin, mais s'il s'agit d'un démon qui tourne en permanence, cette fuite mémoire n'est pas forcément à négliger.

La solution pour éviter cela est de "casser" la circularité avant de modifier la valeur de la variable `$r` :

```
$r->[1] = undef;
```

Je viens de casser le lien indiqué par un astérisque, il n'y a plus de boucle dans les références. Maintenant et seulement maintenant, je puis sans risque écrire :

```
$r = undef;
```

Et la mémoire sera libérée...

9.12 Références sur fichiers

Une ouverture de fichier crée une variable dont il est possible de prendre l'adresse. Voici la syntaxe pour cela :

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

La variable scalaire `$reff` est une référence vers le descripteur de fichier `FILE`. Il nous est aussi possible de créer une référence vers un des trois fichiers préexistants :

```
my $refo = \*STDOUT;
```

Voici des exemples d'utilisation de ces références :

```
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 );
```

Cette dernière ligne est équivalente à `close(FILE);` On peut sans restrictions stocker une référence vers fichier dans une table de hachage ou dans un tableau.

9.13 Références sur fonctions

Comme dans le langage C, une fonction a elle aussi une adresse mémoire. Cette adresse correspond à l'endroit de la mémoire où le code de la fonction est stocké. On peut obtenir une référence vers une fonction de la manière suivante :

```
sub affcoucou {
   my ($p) = @_;
   print "Coucou $p\n";
}
my $ref = \&affcoucou;
```

La variable scalaire `$ref` est une référence vers la fonction `affcoucou`. Elle peut s'utiliser des façons suivantes :

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

Notez bien qu'il est tout à fait possible de stocker une telle référence dans un tableau ou dans une table de hachage...

9.14 Un dernier mot sur les références

De telles structures nous donnent envie de faire de la programmation objet (champs et méthodes pourraient être stockés dans une table de hachage). N'allons pas trop vite, car Perl a été prévu pour la programmation objet et propose les fonctionnalités nécessaires (héritage...) en se basant sur les modules.

Chapitre 10

Les modules

Nous allons aborder ici l'usage et l'écriture de modules, c'est-à-dire de bibliothèques ou encore librairies. Perl tire sa puissance de la richesse des modules existants ; peu d'autres langages (voire aucun) ne peuvent prétendre être aussi riches que Perl. Par exemple, quasiment tous les protocoles réseau auxquels vous pouvez penser sont accessibles en Perl en utilisant un module existant.

En quelques mots, un module est un ensemble de fonctions regroupées, dans un fichier. Ces fonctions y sont regroupées car elles touchent toutes à un même domaine, à un même ensemble de fonctionnalités autour d'une même utilisation, d'un même protocole...

La première chose que je vous invite à faire, c'est de lancer la commande `perl -V` : elle affiche toutes sortes d'informations, dont le contenu de la variable `@INC`. Cette variable de type tableau contient la liste des répertoires où seront recherchés les modules. Le nom `INC` rappelle étrangement (et c'est voulu) la notion d'*include* en C. L'ordre des répertoires de cette variable est important car si un module vient à être présent dans deux répertoires, seule l'occurrence présente dans le premier répertoire de la liste comptera (mais ce cas proviendrait plutôt d'une erreur de nommage ou d'installation).

10.1 Utilisation d'un premier module

Il existe de nombreux modules déjà installés sur votre système ; une distribution de Perl inclut les modules les plus utilisés.

Ainsi, le module nommé `Math::Trig` ; permet d'accéder à des fonctions mathématiques de trigonométrie autres que les seuls cosinus et sinus prédéfinis dans Perl sous les noms de `cos` et `sin`.

Je vous invite à taper la commande `perldoc Math::Trig` dans un terminal ; vous visualiserez ainsi la documentation de ce module. Cette commande `perldoc` fonctionne un peu comme la commande `man` ; vous verrez détaillées les fonctionnalités auxquelles vous avez accès avec un module donné. Vous taperez « q » pour quitter.

On voit ici que nous avons accès à des fonctions comme `tan`, `acos` ou `asin` ainsi qu'à des fonctions de conversion entre unités d'angles ou bien à la valeur de pi. De petits exemples simples d'utilisations vous sont aussi fournis.

La première ligne de code à écrire pour utiliser un module est la suivante :

```
use NomDuModule;
```

Dans bien des cas, cette instruction ajoute des fonctions et des variables à l'espace de nommage (nous reviendrons sur ce point dans la suite). Pour notre exemple, la ligne est :

```
use Math::Trig;
```

Cette ligne devra être placée dans chaque script qui fait usage du module et être exécutée avant tout usage de fonctions ou de variables du module. Typiquement toutes les lignes `use` sont regroupées au début du script.

Vous remarquerez que la ligne `use strict;` que je vous ai conseillé de placer dans chaque script, n'est en fait que le chargement d'un module ; ce module ayant pour rôle de rendre la syntaxe Perl plus coercitive. Le nom des modules de ce type est en minuscules. Ils sont appelés modules pragmatiques. Ils ont pour objet de modifier ou d'étendre la sémantique de Perl. Ainsi `diagnostics` permet d'avoir des messages d'erreurs plus complets (vous pouvez charger sur CPAN la version 1.2-alpha1 qui vous permet d'avoir ces messages avec des explications en français).

Revenons à notre module `Math::Trig`. Voici un exemple de code Perl l'utilisant (j'en plagie ici la documentation) :

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

Je laisse au lecteur le soin de deviner (ou plutôt comprendre) ce que font ces instructions. Une fois chargé, un module n'est pas "déchargeable".

10.2 D'autres modules

Voici l'exemple d'un autre module : `File::Copy` ; il permet certaines manipulations de fichiers lourdes à mettre en œuvre avec de simples appels système. Il est par exemple possible de copier un fichier vers un autre (fichiers disque ou flux de données), ou d'en déplacer un d'une partition vers une autre (impossible avec l'appel système `rename`). Comme l'indique `perldoc File::Copy` :

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

Voici un autre exemple de module en action. Il s'agit du module `Net:FTP` qui nous permet d'accéder très simplement aux fonctionnalités d'un client FTP. Voici, par exemple, comment se connecter sur un serveur (en mode passif, car j'ai un firewall), changer de répertoire et télécharger un fichier :

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

```
use warnings;
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();
```

On remarquera au passage la notation objet (`new`, `->`) ; beaucoup de modules l'utilisent. Même si nous ne verrons la programmation objet que dans une prochaine partie, il est aisé de comprendre comment utiliser de tels modules (de toute façon la documentation des modules comporte des exemples).

Quoi de plus simple finalement pour faire du FTP client ? Comment faisais-je avant pour mettre à jour mon site web des seules pages que j'ai modifiées depuis la dernière fois ? ? ?

10.3 Où trouver les modules ?

C'est très bien tout cela, mais comment trouver le module qui répond à mon problème ? Pour cela je dois vous présenter l'archive de tous les modules Perl, j'ai nommé CPAN (Comprehensive Perl Archive Network). Cette archive recense tous les modules diffusés pour Perl.

Je vous invite à visiter le site `http://www.cpan.org/` vous y trouverez de tout à propos de Perl. Vous pouvez télécharger les sources de l'interpréteur (*Perl source code*), des versions compilées (*Perl binary distributions*) disponibles pour de très nombreuses plates-formes, ainsi que de la documentation sur les modules et de quoi les télécharger. Le lien intéressant est : *CPAN modules, distributions, and authors (search.cpan.org)*. Une page vous est proposée avec de nombreuses sections listant des modules regroupés par thème, ainsi qu'un champ de saisie servant à la recherche de mots clefs dans les modules. Je vous invite à entrer *SMTP* et à voir la variété des modules qui gravitent autour de ce protocole ; le module le plus intéressant est sûrement `Net::SMTP` (plus le nom d'un module est court et semble canonique, plus il y a de chances qu'il soit intéressant). Différents liens permettent de visualiser la documentation (le `perldoc` correspondant) ainsi que de télécharger le module le cas échéant.

L'installation de modules CPAN n'est pas au menu de ce document, je ne vais pas m'appesantir sur la question ; juste deux mots pour vous dire que les commandes `perl~Makefile.PL`, `make`, `make~test` et `make~install` sont la clef du succès. Les modules CPAN sont présents sous forme de package dans toute bonne distribution Linux. Par exemple sous Debian, il s'agit des paquets `libxxx-yyy-perl` (où `xxx-yyy` correspond au nom du module `Xxx::Yyy` mis en minuscules).

Même si vous n'installez pas de module, CPAN n'en reste pas moins la source majeure d'informations sur les modules de votre système. Vous vous rendez par exemple compte que le module `Net::SMTP` répond à vos besoins, vous vérifiez alors que ce module est présent sur votre système en tapant `perl -e 'use  Net::SMTP'` et vous n'avez plus qu'à l'utiliser. La documentation sera accessible par `perldoc Net::SMTP`

Juste pour sourire deux minutes, je vous invite à rechercher dans CPAN un module nommé `Sex` écrit un premier avril et d'en lire la documentation (ainsi que le code)...

10.4 Écrire un premier module

Après avoir manipulé des modules existants et avoir parlé de CPAN, nous allons maintenant apprendre à écrire nos propres modules.

Pour écrire un module, nous devons 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` ; pour commencer vous pourriez le placer dans votre répertoire de travail à côté du script qui l'utilisera, car le répertoire `.` est présent dans ce tableau `@INC`.

Ce fichier doit contenir une première ligne indiquant le nom du module ; pour cela, vous devez écrire :

```
package Utils;
```

Il est important de voir que le nom du package doit être le même que celui du fichier (à l'extension près). Le fichier peut ensuite contenir des définitions de fonctions. Voici un exemple simple d'un tout petit module complet :

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

Il est important de ne pas oublier la dernière ligne, celle qui contient `1;` ; nous reviendrons plus tard sur son rôle.

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

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

La dernière ligne correspond à l'appel de la fonction `bonjour` du module `Utils`. La syntaxe est la suivante : le nom du module est suivi de deux signes deux-points puis du nom de la fonction.

10.5 Et les variables ?

Il est possible de déclarer des variables propres au module. Ces variables seront :

- soit accessibles exclusivement aux fonctions présentes dans le module (on pourrait parler de variables privées, correspondant aux variables `static` déclarées en dehors des fonctions en C) ;
- soit aussi accessibles à l'extérieur du module (on pourrait parler de variables publiques ou variables globales).

Une variable accessible exclusivement aux fonctions du module se déclare avec `my` (que l'on connaît déjà). Une variable aussi accessible depuis l'extérieur du module se déclare avec `our` (`my` pour dire "à moi", `our` pour dire "à nous"). Avant la version 5.6 de Perl, on utilisait un autre mécanisme dont je ne parlerai pas ici, assurez-vous d'avoir une version récente de Perl (cf `perl -v`). Ces variables doivent être déclarées en dehors de toute fonction ; les variables déclarées dans les fonctions sont comme toujours locales au bloc dans lequel elles sont déclarées.

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

Que la variable soit déclarée avec `my` ou avec `our`, il est tout à fait possible d'y accéder depuis une fonction du module (ici `bonjour`). À l'inverse, depuis l'extérieur du module, c'est-à-dire depuis le script, seule la variable `$x` est accessible.

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

De même que pour les fonctions, les noms de variable sont préfixés par le nom du module puis deux signes deux-points. Ici, le nom complet de la variable est donc `Utils::x` qu'il faut faire précéder d'un signe dollar, ce qui donne : `$Utils::x` au final. Il n'y a pas d'erreur de ma part : on n'écrit pas `Utils::$x` ! `:-)`

10.6 De la dernière ligne d'un module

Jusqu'ici, nous avons toujours placé une ligne `1;` à la fin du fichier de notre module. Il faut savoir que cette valeur est la valeur du chargement du module (valeur de l'instruction `use Utils;`) et qu'elle indique si ce chargement s'est bien passé ou non : une valeur fausse indique un problème, une valeur vraie (comme ici `1`) indique au contraire que le chargement s'est bien déroulé. Une valeur fausse mettra fin au script qui fait appel à l'instruction `use`.

Il est donc tout à fait possible de mettre une autre valeur qu'une valeur constante ; on peut, par exemple, envisager mettre un test en dernière instruction pour vérifier si les conditions sont réunies pour l'usage du module. On pourrait imaginer conditionner le chargement du module à l'ouverture d'un fichier, d'une connexion réseau (ou je-ne-sais-quoi encore...). Je ne donnerai pas d'exemple ici, car le cas est rare de la nécessité d'un tel usage, mais il faut savoir que cela est possible.

10.7 Répertoires

Voyons maintenant comment créer des modules aux noms composés comme `Truc::Utils` (nous avons par exemple vu le module `Net::FTP`). Ces noms composés permettent de regrouper les modules par type d'usages ; par exemple `Net` correspond à tout ce qui concerne le réseau.

Revenons à notre exemple `Truc::Utils`. Ce nom `Truc` correspond à un répertoire qui doit être présent dans un des répertoires de la variable `@INC` (par exemple `.`) et le fichier `Utils.pm` doit être présent dans ce répertoire `Truc`.

Voici un exemple de tel module :

```
# --- fichier Truc/Utils.pm ---
package Truc::Utils;
use strict;
use warnings;
our $x = 'toto';
sub bonjour {
   my ($prenom) = @_;
   print "Bonjour $prenom\n";
}
1;
```

Et voici un script l'utilisant :

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

Rien de sorcier.

10.8 Blocs BEGIN et END

Les amoureux de `awk` retrouveront ici deux de leurs enfants préférés `:-))`. 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`).

```
package Utils;
use strict;
use warnings;
sub f {
   ...
}
BEGIN {
   print "Chargement du module\n";
}
END {
   print "Fin d'usage du module\n";
}
1;
```

Notez bien qu'il ne s'agit pas de fonctions (pas de mot clef `sub`), mais bien de blocs labélisés. Le bloc `BEGIN` sera exécuté lors de l'instruction `use Utils;` avant toute autre instruction du module (y compris les `use` placés dans le module). Le bloc `END` sera exécuté lors de la fin du programme.

L'usage de ces deux blocs peut être nécessaire lorsque l'utilisation du module est conditionnée par l'obtention d'une ou plusieurs ressources comme un fichier ou une connexion réseau. Ces blocs vont nous servir à préparer le terrain au début et à libérer les ressources à la fin.

Lorsque dans un module sont présentes d'autres instructions que des définitions de variables et des définitions de fonctions, ces instructions sont exécutées au moment du chargement du module. Tout se passe comme si ces instructions figuraient dans un BEGIN implicite. L'usage d'un bloc BEGIN permet juste au programmeur d'écrire un code un peu plus lisible et propre, dans la mesure où toutes ces instructions sont regroupées sous un nom (BEGIN) qui rappelle explicitement qu'elles sont exécutées au début.

Notez bien que la programmation objet (lire la suite) a quelque peu rendu ces deux blocs obsolètes, voire inutiles.

10.9 Introduction à l'export de symboles

Sous ce titre barbare se cache une idée simple : il peut être pénible de toujours écrire le nom complet des fonctions de modules. Je veux dire par là que d'écrire `Utils::bonjour` à chaque fois que vous voulez appeler cette fonction est sans doute lourd et est quelque peu pénible à la longue. Il existe un moyen pour n'avoir qu'à écrire `bonjour` sans avoir à rappeler le nom du module qui contient la fonction. En faisant cela, nous allons ajouter la fonction dans l'espace de nommage du script.

Placer une fonction ou une variable d'un module dans l'espace de nommage d'un script ou d'un autre module s'appelle faire un *export*, on parle d'exporter le symbole (fonction ou variable). Ce symbole est donc importé par le script.

Pour avoir la capacité d'exporter des symboles, notre module futur-exportateur doit comporter les lignes suivantes :

```
package Utils;
use Exporter;
our @ISA = qw(Exporter);
```

Ces deux nouvelles lignes d'instructions doivent être placées juste après l'instruction `package`. La première est l'invocation du module `Exporter` ; avec la seconde on indique que notre module est un (`ISA`) `Exporter` (nous reverrons cette syntaxe et ses implications en programmation objet). Notre module est maintenant capable d'exporter des symboles.

Il existe quatre types de symboles (fonctions ou variables) :

- ceux qui sont exportés pas défaut : le script utilisant le module n'a besoin de rien faire de spécial (autre que de faire le `use`) pour que ces symboles soient exportés,
- ceux qui sont individuellement exportables en fonction de ce que demande le script utilisateur ;
- ceux qui sont exportables en groupe (on parle de *tags*) selon ce que demande le script utilisateur ;
- ceux qui ne sont pas exportables (c'est-à-dire qu'il faudra toujours faire précéder leur nom par le nom complet du module).

Chacun des trois premiers ensembles est associé à une variable déclarée avec `our`.

10.10 Export par défaut de symboles

Les symboles exportés par défaut doivent être listés dans la variable `@EXPORT` ; il s'agit donc d'un tableau. Il est courant d'initialiser ce tableau avec l'opérateur `qw` que nous avons déjà vu et sur lequel je ne reviendrai donc pas :

```
our @EXPORT = qw(&bonjour &hello $var);
```

Cette ligne placée dans le module `Utils` à la suite de la ligne

```
our @ISA = qw(Exporter);
```

va permettre d'utiliser les fonctions `bonjour` et `hello` ainsi que la variable scalaire `$var` sans préfixe dans le script utilisateur. Notez bien que, si les variables doivent être citées avec leur caractère de différentiation de type (le dollar, l'arobase ou le pourcentage), il en est de même avec les fonctions et le signe « esperluette » (`&`). Sachez juste que cet esperluette peut être omis.

Voici comment on peut maintenant utiliser le module en question :

```
use Utils;
bonjour("Paul");
hello("Peter");
print "$var\n";
```

Plutôt simple.

10.11 Export individuel de symboles

Un symbole peut être exporté à la demande de celui qui utilise le module. C'est-à-dire que ce symbole n'est pas exporté par défaut, mais il peut faire l'objet d'un export s'il est nommément cité lors de l'instruction `use`.

Un symbole doit apparaître dans la variable `@EXPORT_OK` pour être autorisé à être exporté :

```
our @EXPORT_OK = qw(&gutenTag &ciao $var2);
```

Ces trois symboles sont maintenant exportables dans le script utilisant ce module. Pour cela il convient d'ajouter une liste de symboles à l'instruction `use` :

```
use Utils qw(&ciao $var2);
ciao("Paula");
print "$var2\n";
```

Cette ligne importe donc les symboles demandés et ces derniers sont donc utilisables sans préfixe.

Il se trouve qu'une telle ligne n'importe plus les symboles par défaut (ceux de la variable `@EXPORT`) ; ne me demandez pas pourquoi, je trouve cela aussi stupide que vous... Pour remédier à cela, il nous faut ajouter à la liste des imports le tag `:DEFAULT` :

```
use Utils qw(:DEFAULT &ciao $var2);
bonjour("Paul");
hello("Peter");
print "$var\n";
ciao("Paula");
print "$var2\n";
```

Ce mécanisme de sélection des symboles exportés permet à l'utilisateur du module de ne pas trop polluer son espace de nommage et de choisir les seules fonctions dont il aura besoin.

10.12 Export par tags de symboles

Il est possible de regrouper les symboles dans des tags. Un tag est une liste de symboles. L'import d'un tag provoque l'import de tous les symboles composant ce tag. La variable qui entre ici en jeu est `%EXPORT_TAGS` ; il s'agit donc d'une table de hachage. À chaque tag est associée une référence vers un tableau contenant la liste des symboles du tag :

```
our %EXPORT_TAGS=(T1=>[qw(&ciao &gutenTag)],
                  T2=>[qw(&ciao $var2)]);
```

Le tag `T1` est associé aux fonctions `ciao` et `gutenTag`. Le tag `T2` est associé à la fonction `ciao` et à la variable `$var2`. Le nom des tags est par convention en majuscules.

Remarque importante : les symboles présents dans les listes associées aux tags doivent absolument être présents dans `@EXPORT` et/ou `@EXPORT_OK`. Dans le cas contraire, leur export sera impossible.

Voici un usage de ce module :

```
use Utils qw(:T2);
ciao("Paula");
print "$var2\n";
```

Le nom du tag est placé dans la liste des modules, précédé par le signe deux-points. Il est possible de combiner les différents types d'accès :

```
use Utils qw(:DEFAULT &ciao :T1);
bonjour("Paul");
hello("Peter");
print "$var\n";
ciao("Paula");
print "$var2\n";
gutenTag("Hans");
```

On voit alors que `DEFAULT` est un tag.

10.13 Exemple complet d'exports

Voici un exemple complet d'usage des exports dans les modules ; j'ai essayé de regrouper toutes les configurations.

Voici le module `Utils` dans le fichier `Utils.pm` :

```
package Utils;
use strict;
use warnings;
use Exporter;
our @ISA = qw(Exporter);
our @EXPORT = qw(&f1 &f2);
our @EXPORT_OK = qw(&f3 &f4 &f5 &f6);
our %EXPORT_TAGS = (T1 => [qw(&f5 &f6)],
                    T2 => [qw(&f4 &f6)]);
sub f1 { print "f1\n"; }
sub f2 { print "f2\n"; }
sub f3 { print "f3\n"; }
sub f4 { print "f4\n"; }
sub f5 { print "f5\n"; }
sub f6 { print "f6\n"; }
1;
```

Et voici un script l'utilisant (après chaque appel de fonction est signalée la raison qui fait qu'il est possible de l'appeler sans préfixe) :

```
#!/usr/bin/perl
use strict;
use warnings;
use Utils qw(:DEFAULT :T2 &f3);
```



```
f1();         # tag DEFAULT
f2();         # tag DEFAULT
f3();         # individuellement
f4();         # tag T2
Utils::f5();  # pas importée
f6();         # tag T2
```

Notez bien le cas de la fonction `f5` qui n'est pas importée, mais qui n'en reste pas moins utilisable.

Pour plus d'informations et d'exemples je vous invite à vous référer à `perldoc Exporter`.

10.14 Fonctions inaccessibles

Vous allez me poser la question suivante : comment faire pour rendre une fonction d'un module inaccessible depuis le script ? Et je vous répondrai : cela n'est a priori pas possible (vous allez voir que si finalement).

Vous l'avez compris, Perl n'est pas un langage extrêmement coercitif, notamment par rapport à des langages comme Java ou C++. Il n'y a donc rien de prévu dans le langage pour rendre certaines fonctions uniquement accessibles depuis l'intérieur du module. Est alors apparue la convention suivante : toute fonction ou variable dont le nom commence par un souligné (ou *under-score* '_') est privée et ne doit pas être utilisée à l'extérieur du module.

Cette confiance en l'utilisateur du module est souvent suffisante et les modules CPAN sont bâtis sur ce modèle.

Néanmoins, si vous êtes outré par ce que je viens d'écrire, car vous êtes un fanatique de Java ou autres, il existe un moyen d'écrire des fonctions vraiment internes aux modules. Il faut pour cela déclarer une variable avec `my` (donc invisible depuis l'extérieur du module) et d'en faire une référence anonyme vers fonction :

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

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

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

Remarquez que, comme le code Perl est toujours accessible en lecture, il est toujours possible à l'utilisateur du module de prendre le code en copier-coller et d'en faire une fonction personnelle... Perl n'est pas fait pour les paranoïaques.

10.15 Documentation des modules

Documenter son travail est important pour une réutilisation du code par d'autres ou même par soi-même plus tard... 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 (=).

La documentation d'un module commence typiquement ainsi :

```
=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 en-têtes 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.

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. Pour cela vous devez savoir que 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 peut alors reprendre.

```
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";
}
```

Comment visualiser une telle documentation, allez-vous me demander ? Rien de plus simple : `perldoc` est notre allié ! Tapez donc `perldoc Utils` (ou tout autre nom que vous aurez choisi de donner à votre module) et sa documentation apparaît au format `man` comme tout bon module CPAN. Quoi de plus simple ?

```
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.
```

Pour plus d'informations et de détails sur ce format, je vous invite à consulter `perldoc perlpod` où de nombreux exemples sont donnés. Vous pouvez aussi jeter un œil au code d'un module ou deux dont le `perldoc` vous intrigue...

10.16 Un dernier mot sur les modules

J'espère que la lecture de cette partie vous a donné envie de structurer votre code en regroupant vos fonctions dans de telles bibliothèques. Vous vous rendrez compte de cette nécessité lorsque vous aurez un fichier de code trop gros à gérer... Mais même sans cela, n'hésitez pas à faire des modules, ne serait-ce que pour une réutilisation de fonctionnalités dans d'autres scripts ou pour partager votre code avec des amis, des collègues, voire avec la communauté.

Chapitre 11

Programmation objet

La programmation objet est le concept nouveau de ces vingt dernières années. C++ est bâti sur le C et apporte l'objet. Java a été mis au point (entre autres) pour passer outre les nombreux pièges et problèmes de C++. Ruby est un langage interprété basé sur ce concept objet.

Perl, qui utilisait un garbage collector bien avant que Java n'existe, ne pouvait pas être en reste et la communauté Perl a rapidement proposé les extensions du langage nécessaires à ce type de programmation.

On notera que ces extensions sont peu nombreuses, car l'idée a été de réutiliser au maximum ce qui existait déjà en Perl et de l'appliquer à la programmation objet. Le C++ étant compilé et devant rester compatible avec le C, cela fut un challenge de mettre sur pied ce langage ; cela explique sans doute pourquoi C++ est si complexe et comporte tant de pièges. Perl, de par sa nature interprétée, n'a pas posé de problème pour s'étendre à l'objet.

La programmation par objets ouvre le champ des possibilités offertes au programmeur ; alliée à un langage puissant et flexible comme Perl, elle offre la souplesse, la richesse et la facilité d'écriture qu'il manque aux langages uniquement objet. Toutefois, de par sa nature permissive, le langage Perl ne saurait être aussi strict que des langages exclusivement objet. Le programmeur est invité à faire les choses proprement, mais rien ne l'y oblige.

11.1 Vous avez dit objet ?

Sans revenir sur la théorie de la programmation objet, je vais tenter ici d'y faire une courte introduction. La programmation orientée objet est un type de programmation qui se concentre principalement sur les données. La question qui se pose en programmation OO (orientée objet) est "quelles sont les données du problème ?" à l'instar de la programmation procédurale par exemple, qui pose la question « quelles sont les fonctions/actions à faire ?nbsp ;». En programmation OO, on parle ainsi d'objets, auxquels on peut affecter des variables/attributs (propriétés) et des fonctions/actions (méthodes).

On parle de « classenbsp ;», qui est une manière de représenter des données et comporte des traitements : une classe « Chaussurenbsp ;» décrit, par exemple, les caractéristiques d'une chaussure. Elle contient un champ décrivant la pointure, la couleur, la matière, etc. Une telle classe comporte de plus des traitements sur ces données ; ces traitements sont appelés « méthodesnbsp ;». Grossièrement une méthode est une fonction appliquée à un objet.

Une fois définie une telle classe, il est possible d'en construire des instances : une instance d'une classe est dite être un objet de cette classe. Dans notre exemple, il s'agirait d'une chaussure

dont la pointure, la couleur et la matière sont renseignées.

11.2 Préparatifs

Nous allons maintenant voir comment écrire une classe en Perl. Vous verrez, cela est très simple et démystifie la programmation objet.

En Perl, une classe n'est autre qu'un module et un objet (instance de cette classe) n'est autre qu'une référence associée à cette classe. Dans le constructeur, nous allons donc créer une référence (typiquement vers une table de hachage) et nous allons l'associer au package en question ; lors de cette association, on dit en Perl 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 clef pour le nom du champ et de la valeur pour la valeur du champ.

Voyons un exemple : définissons une classe `Vehicule` qui comporte deux champs : un nombre de roues et une couleur.

11.3 Écrire un constructeur

Nous définissons un package `Vehicule` dans un fichier `Vehicule.pm` comme nous le faisons pour tout module.

```
 1:  # --- fichier Vehicule.pm ---
 2:  package Vehicule;
 3:  use strict;
 4:  use warnings;
 5:  sub new {
 6:     my ($class,$nbRoues,$couleur) = @_;
 7:     my $this = {};
 8:     bless($this, $class);
 9:     $this->{NB_ROUES} = $nbRoues;
10:     $this->{COULEUR} = $couleur;
11:     return $this;
12:  }
13:  1;  # À ne pas oublier...
```

La ligne numéro 2 indique le nom du package actuel ; il est conseillé de choisir le même nom que pour le fichier, simplement pour des raisons d'organisation et de maintenance. La ligne 12 comporte le fameux code de retour du chargement du module. La ligne 3 force une syntaxe plus rigoureuse. En Perl, le nom des modules et donc des classes que le programmeur définit doit être composé de majuscules et de minuscules, avec typiquement une majuscule au début de chaque mot ; les noms de package exclusivement en minuscules sont réservés pour les modules pragmatiques de Perl (dits modules *pragma* comme `strict`, etc.), les noms exclusivement en majuscules sont inélégants. `:-)`

Nous définissons une fonction `new` (ligne 5) dont le but est de construire un objet `Vehicule`. Il s'agit donc d'un constructeur ; un constructeur en Perl est une simple fonction renvoyant un objet. Il est bon de noter que le choix du nom pour cette fonction est totalement libre ; il est courant de l'appeler `new`, mais rien ne nous y oblige. On pourrait par exemple choisir le nom de

la classe (si on est un habitué de C++ ou de Java), mais on verra par la suite que cela n'est pas forcément une bonne idée.

Cette fonction prend en premier paramètre le nom de la classe ; cela semble superflu, mais on verra qu'il n'en est rien. Les paramètres suivants sont laissés à la discrétion du programmeur : bien souvent on passe ici les valeurs de champs pour l'initialisation. C'est ce que l'on fait ici : le nombre de roues et la couleur sont transmis (ligne 6).

Ligne 7, nous créons une référence anonyme vers une table de hachage vide `{}`. Cette référence est stockée dans une variable scalaire nommée `$this`, car il va s'agir de notre futur objet. Le nom de cette variable est totalement arbitraire et j'ai choisi de prendre le nom `$this` car il rappelle les variables `this` de C++ et de Java. Mais comprenez bien qu'il n'y a rien d'obligatoire dans cette appellation. Il est d'ailleurs fréquent que le nom de l'objet dans les méthodes soit plutôt `$self` en Perl (vous trouverez ce nom dans la plupart des modules objet de CPAN).

Ligne 8, nous indiquons que cette référence est liée au package (à la classe) `$class`. Cette variable `$class` vaudra ici `Vehicule`. L'opérateur `bless` associe le package en question à la référence. La référence est maintenant liée au package.

Dans les lignes 9 et 10, 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 qui constitue l'objet. Pour affecter un champ de l'objet que nous sommes en train de construire, il suffit de créer un couple clef/valeur dans la table de hachage référencée par `$this`. J'ai pris l'habitude de mettre le nom des champs en lettres majuscules.

Notez que le nombre, le nom et le contenu des champs peuvent donc varier d'une instance de la classe à une autre instance de cette même classe. Libre au programmeur de faire ce qu'il veut : si le but est de vraiment programmer objet de façon formelle, il va respecter les habitudes de ce type de programmation qui veut que toutes les instances d'une même classe aient les mêmes champs ; mais s'il ne tient pas à respecter ces contraintes, il est libre de faire ce qu'il veut de chacun de ses objets.

La ligne 11 consiste en un `return` de la référence vers la table de hachage ainsi construite.

11.4 Appeler le constructeur

Pour faire usage de cette classe, nous allons devoir disposer d'un script écrit dans un autre fichier (par exemple `script.pl`) :

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

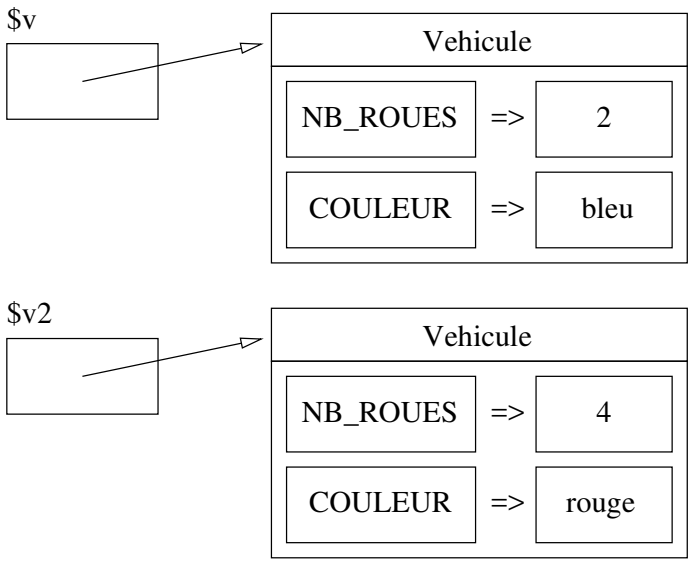
Comme pour tout module, nous devons explicitement indiquer que nous allons l'utiliser :

```
use Vehicule;
```

Nous pouvons maintenant utiliser le constructeur que nous avons défini :

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

Nous venons ici de créer deux instances de la classe `Vehicule`. On dit que ces deux variables `$v` et `$v2` sont des `Vehicule`. Ces deux objets sont donc indépendants l'un de l'autre ; ils sont de la même classe `Vehicule`, mais en constituent des instances autonomes, la modification de l'un ne modifiant pas l'autre. Voici un schéma de l'état de notre mémoire :



Cette syntaxe `Vehicule->new` correspond à l'appel du constructeur `new` que nous venons d'écrire. La variable `$v` est initialisée à la valeur de retour de cette fonction. Elle est donc une référence vers une table de hachage dont deux champs sont initialisés et qui a été bénie (`bless`) en `Vehicule`. Idem pour `$v2`.

Il est aussi possible de faire usage de la syntaxe suivante :

```
my $v = new Vehicule( 2, "bleu" );
```

Cette formulation va sans doute rassurer les habitués de Java ou de C++, mais peut induire le programmeur en erreur. En effet, cette dernière syntaxe semble indiquer que `new` est un opérateur spécifique pour appeler un constructeur et est donc un mot réservé du langage. Il n'en est rien ; comme on le verra un peu plus loin, ce nom est totalement arbitraire.

11.5 Manipulations de l'objet

Revenons à notre exemple. En plus de savoir qu'elle pointe vers une table de hachage, la référence `$v` sait de quelle classe elle est. En effet, si nous l'affichons :

```
print "$v\n";
```

Nous obtenons la chose suivante à l'écran :

```
Vehicule=HASH(0x80f606c)
```

Je vous rappelle que dans le cas de l'affichage d'une référence vers une table de hachage non bénie, nous obtenons quelque chose de la forme :

```
HASH(0x80fef74)
```

Un objet (à partir de maintenant, nommons ainsi une référence vers une table de hachage bénie) sait donc de quelle classe il est. Cela va lui permettre de choisir le bon package quand on appellera une méthode sur cet objet (lire la suite).

Voyons maintenant ce que donne le module `Data::Dumper` (dont j'ai déjà parlé) sur une telle référence :

```
use Data::Dumper;
print Dumper($v)."\n";
```

L'affichage suivant est effectué :

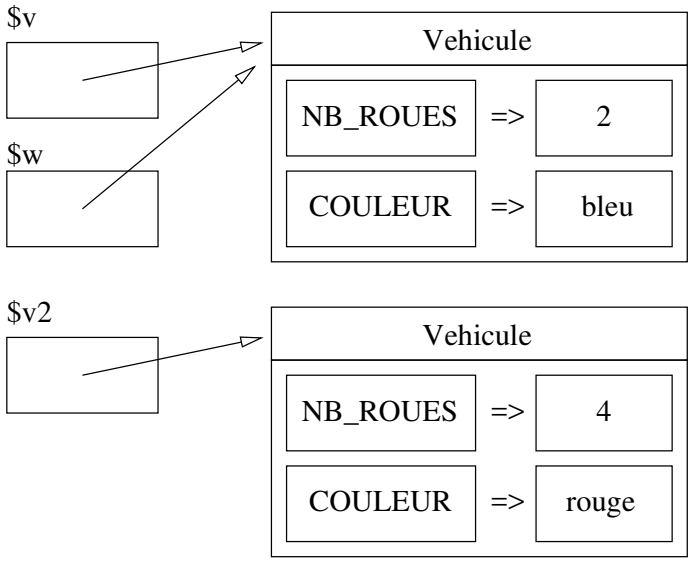
```
$VAR1 = bless( {
                 'COULEUR' => 'bleu',
                 'NB_ROUES' => 2
               }, 'Vehicule' );
```

On remarquera que la tradition de `Data::Dumper` qui consiste en ce que la chaîne renvoyée est directement intégrable dans un code Perl est respectée : même l'opération de bénédiction (`bless`) est présente.

Il faut bien voir que `$v` est une référence vers un objet. Autrement dit, si on fait une copie de cette variable

```
my $w = $v;
```

nous obtenons deux variables qui pointent vers le même objet.



La modification de l'objet pointé par l'un modifiera celui pointé par l'autre, car il s'agit du même objet.

11.6 Plusieurs constructeurs

Comment écrire plusieurs constructeurs pour une même classe ? Rien de plus simple. Le nom `new` que j'ai choisi pour le constructeur n'a rien d'obligatoire, en fait un constructeur est une simple fonction qui renvoie une référence bénie. Seule la syntaxe d'appel change par rapport à ce que l'on a vu pour les modules. Je peux donc écrire un autre constructeur (donc avec un autre nom) :

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

Ce constructeur de `Vehicule` prend par exemple un seul paramètre (la couleur) et n'affecte pas de champs `NB_ROUES`, car rien de ne l'y oblige. À moi de savoir comment je veux gérer mes véhicules, à savoir comment j'autorise qu'ils soient construits.

Pour être propre et cohérent, nous allons tout de même considérer qu'il est sage d'affecter une valeur à la clef `NB_ROUES` :

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

Voici comment on va faire appel à ce constructeur :

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

De la même façon que pour le précédent constructeur, il est possible d'utiliser la syntaxe suivante :

```
my $v2 = nouveau Vehicule( "bleu" );
```

Ce qui est, reconnaissons-le, quelque peu déroutant. C'est pour cela que je vous conseille d'utiliser plutôt la première syntaxe `Vehicule->nouveau()` ou alors de vous en tenir à un constructeur `new` pour écrire `new Vehicule()`.

11.7 Écrire une méthode

Une méthode est une fonction qui s'applique à une instance de la classe. Cela est vrai dans tous les langages objet, mais bien souvent cela est masqué ; dans notre cas rien de masqué, le premier paramètre de la fonction sera l'objet (la référence bénie) :


```
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. Vous noterez une fois de plus que rien n'oblige le programmeur à nommer cette variable `$this` ; la seule contrainte est sa première place parmi les arguments.

Cette méthode peut dès maintenant être appelée depuis le fichier `script.pl` sur les objets de type `Vehicule`. Il n'y pas de nécessité de faire usage du lourd mécanisme d'`Exporter`, car nous n'avons pas besoin de modifier l'espace de nom des fonctions des scripts appelants.

Pour appeler la méthode, il suffit d'écrire dans le fichier `script.pl` :

```
$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.

L'affichage suivant a donc lieu :

```
Avec 2 roues, je roule à 15.
```

Écrivons par exemple une méthode d'affichage :

```
sub toString {
   my ($this) = @_;
   return "(Vehicule:$this->{NB_ROUES},$this->{COULEUR})";
}
```

Cette méthode renvoie une chaîne de caractères, représentation de l'objet. Les habitués de Java noteront que j'ai choisi le nom de cette fonction pour leur rappeler des souvenirs, mais qu'il n'a rien de spécial. Voici comment l'utiliser dans le script :

```
print $v->toString()."\n";
```

Et l'affichage a lieu :

```
(Vehicule:2,bleu)
```

Libre à vous de choisir un plus bel affichage.

11.8 Reparlons des champs

Un champ est une donnée propre à une instance de classe. En Perl, ces champs sont stockés comme clef/valeur dans la table de hachage qui constitue l'objet (si on utilise une table de hachage comme objet, ce qui est souvent le cas). Nos véhicules comportent deux champs : `NB_ROUES` et `COULEUR`.

Ces champs 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 dans le fichier `script.pl` :

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

C'est-à-dire que je peux accéder sans restriction à l'ensemble des champs de l'objet. En effet, j'ai en main une référence vers une table de hachage ; le fait qu'elle soit bénie ne change rien au fait que je peux la déréférencer et accéder aux diverses valeurs qu'elle contient. Je peux aussi bien modifier ces valeurs ou même en ajouter ou en supprimer, car il s'agit en effet d'une table de hachage comme les autres.

« Comment protéger les données d'un objet ? », allez-vous alors me demander. Eh bien, Perl n'a rien prévu pour ça. Cela va sans doute faire hurler les puristes de la programmation objet, mais c'est comme cela... Perl vous propose les principaux mécanismes pour faire de la programmation objet tout en restant cohérent avec le reste du langage, certaines choses ne sont donc pas possibles.

Faute de champs privés en Perl, il existe une convention qui dit que les champs dont la clef commence par un underscore (souligné `_`) sont des champs privés et les scripts qui utilisent les objets ainsi faits sont priés de respecter cette convention. C'est le cas de beaucoup de modules CPAN. Cette convention est aussi valable pour les méthodes (une méthode dont le nom commence par un underscore est une méthode privée).

De façon générale, un programmeur Perl est quelqu'un de bonne éducation (sinon il programmerait en Java `;-)`)) et il ne modifiera pas une instance d'une classe qu'il n'a pas écrite. En effet, pourquoi modifier un objet `Net::FTP` alors qu'il fait très bien son travail ? De toute façon, il a forcément accès au code source de cette classe et s'il veut la modifier, il peut en faire une copie et la modifier !

La protection des données est donc plus une nécessité sociale (manque de confiance en l'espèce humaine à laquelle les développeurs prétendent encore faire partie `:-)`)) qu'une nécessité technique. Pas dramatique pour faire de l'objet.

11.9 Composition

Prenons le temps de faire un petit exemple pratique pour illustrer le concept de composition. La composition est le fait qu'un objet est constitué d'autres objets. Par exemple un garage comporte des véhicules.

Je décide qu'un garage aura une taille limite : il s'agira du nombre maximal de véhicules qu'il pourra contenir. Un garage devra donc contenir une liste de véhicules.

Nous devons créer un fichier `Garage.pm` contenant :

```
package Garage;
use strict;
use warnings;
# ... ici les méthodes qui vont suivre
1;
```

Voyons ensuite le constructeur :

```
sub new {
   my ($class,$places) = @_;
```

```
   my $this = {};
   bless($this, $class);
   $this->{PLACES} = $places;
   $this->{VEHICULE} = [];
   return $this;
}
```

Ce constructeur prendra en paramètre le nombre de places disponibles du garage ; cette valeur est enregistrée dans le champ `PLACES`. Le champ `VEHICULE` comportera la liste des véhicules ; pour cela elle est initialisée à la référence anonyme vers une liste vide. La méthode `ajoute` se chargera d'ajouter les véhicules dans la limite du nombre de places :

```
sub ajoute {
   my ($this,$vehicule) = @_;
   if( @{$this->{VEHICULE}} < $this->{PLACES} ) {
      push @{$this->{VEHICULE}}, $vehicule;
      return 1;
   }
   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 (l'expression `@{$this->{VEHICULE}}` est la liste pointée par la référence `$this->{VEHICULE}` ; évaluée en contexte numérique, elle vaut son nombre d'éléments). S'il reste de la place, le véhicule est ajouté (fonction `push`) et elle renvoie vrai, sinon elle renvoie faux.

Voici comment l'utiliser dans le script :

```
use Garage;
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");
```

Écrivons maintenant une méthode d'affichage pour un tel objet :

```
sub toString {
   my ($this) = @_;
   my $s = "{Garage:$this->{PLACES},";
   foreach my $v ( @{$this->{VEHICULE}} ) {
      $s .= $v->toString();
   }
   return $s."}";
}
```

On appelle sur chaque objet `Vehicule` la méthode `toString` de façon à le faire apparaître dans la chaîne que nous allons renvoyer. Voici comment appeler cette méthode dans le script :

```
print $g->toString()."\n";
```

Ce qui donne l'affichage suivant :

```
{Garage:3,(Vehicule:2,bleu)(Vehicule:4,vert)(Vehicule:1,jaune)}
```

On pourrait écrire cette méthode différemment, si on voulait séparer chaque véhicule par une virgule :

```
sub toString {
   my ($this) = @_;
   my $s = "{Garage:$this->{PLACES},";
   $s .= join( ',', map( { $_->toString() }
                         @{$this->{VEHICULE}} ) );
   return $s."}";
}
```

Ce qui donne l'affichage suivant :

```
{Garage:3,(Vehicule:2,bleu),(Vehicule:4,vert),(Vehicule:1,jaune)}
```

Pour ajouter encore une difficulté, je décide de trier les véhicules par nombre de roues croissant :

```
sub toString {
   my ($this) = @_;
   my $s = "{Garage:$this->{PLACES},";
   $s .= join( ',', map( {$_->toString()}
      sort( {$a->{NB_ROUES} <=> $b->{NB_ROUES} }
            @{$this->{VEHICULE}} ) ) );
   return $s."}";
}
```

Ce qui donne l'affichage suivant :

```
{Garage:3,(Vehicule:1,jaune),(Vehicule:2,bleu),(Vehicule:4,vert)}
```

Ces deux derniers exemples vous sont proposés pour que vous vous torturiez un peu les méninges ;-))

11.10 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.

La même chose a lieu dans le cas de la disparition d'une variable locale :

```
if( ... ) {
   my $v3 = Vehicule->new(3,'jaune');
   ...
   ...
}
```

La variable `$v3` cesse d'exister à la fermeture de l'accolade du `if`. L'objet qui a été créé dans le bloc sera donc détruit (sauf si on a fait en sorte qu'une autre variable dont la visibilité dépasse ce bloc pointe aussi vers l'objet).

Cette libération n'est peut-être pas faite en temps réel, mais ce n'est pas au programmeur de s'en occuper.

Il existe une méthode très 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.

```
sub DESTROY {
   my ($this) = @_;
   print "À la casse Vehicule ! ";
   print "($this->{NB_ROUES} $this->{COULEUR})\n";
}
```

Cette méthode doit être définie dans le package de l'objet en question et reçoit en premier argument une référence vers l'objet qui va être détruit.

Cette méthode est très utile pour libérer des ressources (fichier, connexion réseau, etc.) qui ont été allouées lors de la création de l'objet.

11.11 Héritage

L'héritage est un des apports de la programmation objet. Perl dispose de tout ce qu'il faut pour le mettre en œuvre.

Imaginons qu'en plus de véhicules, nous avons besoin de manipuler des vélos et des voitures. Ces objets ont en commun d'avoir un nombre de roues et une couleur. Ils ont des caractéristiques supplémentaires qui leur sont propres ; les vélos ont un nombre de vitesses et les voitures, un nombre de sièges. Ces classes `Velo` et `Voiture` vont donc hériter de la classe `Vehicule`, c'est-à-dire qu'elles vont comporter tous les champs et les méthodes de cette classe.

On dit alors que la classe `Vehicule` est la classe mère ; les classes `Velo` et `Voiture` étant des classes filles. Notez bien que je prends comme exemple deux classes filles et qu'il n'est nullement nécessaire d'avoir deux classes filles pour mettre en œuvre l'héritage, une seule est suffisante.

Voyons le cas de la classe `Velo`, créons pour cela un fichier `Velo.pm` qui contient :

```
package Velo;
use strict;
use warnings;
use Vehicule;
our @ISA = qw(Vehicule);

# ... ici les méthodes qui vont suivre

1;
```

On signale le lien de filiation entre classes au moyen de la variable `@ISA` positionnée à la valeur d'une liste contenant le nom de la classe mère. ISA vient de l'anglais "is a", est un : on dit qu'un vélo est un véhicule. Il hérite donc des champs et méthodes de la classe `Vehicule`.

Définissons-lui un constructeur :

```
sub new {
   my ($class,$couleur,$nbVitesses) = @_;
   my $this = $class->SUPER::new( 2, $couleur );
   $this->{NB_VITESSES} = $nbVitesses;
   return bless($this,$class);
}
```

Ce constructeur prend donc deux paramètres. Il appelle le constructeur de la classe mère (syntaxe `$class->SUPER::new`). Il ajoute un champ `NB_VITESSE` et renvoie une référence bénie en `Velo`. Notez bien qu'aucun appel au constructeur de la classe mère n'est fait par défaut, il faut le faire explicitement dans tous les cas.

Le lecteur aura noté que, comme les champs de la classe mère et de la classe fille sont stockés dans la même table de hachage, il n'y a pas de moyen simple pour faire de surcharge ou de masquage des champs. Les noms des champs devront être minutieusement choisis pour ne pas entrer en conflit les uns avec les autres.

Voyons à présent comment écrire une méthode pour cet objet :

```
sub pedale {
   my ($this,$ici) = @_;
   print "Sur mon vélo $this->{COULEUR} ";
   print "je pédale avec $this->{NB_VITESSES} vitesses";
   print " dans $ici.\n";
}
```

Utilisons maintenant tout cela dans notre script :

```
use Velo;
my $velo = Velo->new('blanc',18);
$velo->pedale('les bois');
$velo->roule(10);
```


Un vélo dispose de la méthode `roule`, car il est aussi un véhicule. L'affichage suivant est effectué :

```
Sur mon vélo blanc je pédale avec 18 vitesses dans les bois.
Avec 2 roues, je roule à 10.
```

Voyons à présent comment afficher un tel objet. Nous laisserons le soin à la classe `Vehicule` d'afficher le vélo comme étant un véhicule et nous n'effectuerons dans la classe `Velo` que l'affichage de ce que nous avons ajouté à la classe mère :

```
sub toString {
   my ($this) = @_;
   my $s = "[Velo:$this->{NB_VITESSES}";
   $s .= $this->SUPER::toString();
   $s .= "]";
}
```

La syntaxe `$this->SUPER::toString()` correspond à l'appel de la méthode `toString` de la classe mère. Nous pouvons maintenant l'appeler dans notre script :

```
print $velo->toString()."\n";
```

L'affichage suivant est effectué :

```
[Velo:18(Vehicule:2,blanc)]
```

Rien de plus simple !

Seule chose pas extrêmement pratique, l'appel au destructeur s'arrête au premier destructeur rencontré. Si dans notre exemple nous définissions une méthode `DESTROY` pour la classe `Velo`, la méthode `DESTROY` de la classe `Vehicule` ne sera pas appelée. Cela peut être gênant si des ressources importantes sont libérées dans cette méthode ; il faut alors l'appeler explicitement. Voici un exemple de méthode `DESTROY` pour la classe `Velo` :

```
sub DESTROY {
   my ($this) = @_;
   $this->SUPER::DESTROY();
   print "Bye bye Velo ! ";
   print "($this->{NB_VITESSES} $this->{NB_ROUES} ".
         "$this->{COULEUR})\n";
}
```

La deuxième ligne de la méthode fait un appel à la méthode de la classe mère.

Pour faire de l'héritage multiple en Perl, rien de plus simple. Vous aurez peut-être noté que la variable `@ISA` est un tableau, elle peut donc contenir plusieurs noms de classe :

```
package Voiture;
our @ISA = qw(Vehicule Danger Pollution);
```

La détermination de la bonne méthode à appeler est effectuée dynamiquement par une recherche en profondeur dans l'arbre d'héritage. Je ne m'étendrai pas plus sur cette question de l'héritage multiple.

11.12 Classes d'un objet

Dans cette partie nous allons voir comment connaître la classe d'un objet ainsi que tester l'appartenance à une classe pour un objet.

Souvenez-vous de l'opérateur `ref` qui, appliqué à une référence, renvoie le type de structure de données vers laquelle elle pointe (scalaire, tableau, table de hachage, etc.). Appliqué à un objet, il renvoie la classe de l'objet :

```
print ref($velo)."\n";
print "Ouf, tout va bien !\n"
   if( ref($velo) eq "Velo" );
```

L'affichage effectué est le suivant :

```
Velo
Ouf, tout va bien !
```

Il s'agit donc de la classe "principale" de l'objet. Sachant qu'un vélo est aussi un véhicule, il peut être utile de pouvoir tester l'appartenance de l'objet à une classe plutôt que de connaître sa classe principale. Par exemple, si nous manipulons un tableau comportant des objets divers et variés et si nous souhaitons y retrouver tous les objets de classe `Vehicule` pour appeler leur méthode `roule`, nous ne pouvons pas écrire

```
if( ref($r) eq "Vehicule" ) { ... }
```

car les vélos ne seront pas sélectionnés. En fait la question que nous devons ici nous poser n'est pas de savoir si la classe principale de l'objet est `Vehicule`, mais nous devons nous demander si l'objet *est un* objet de classe `Vehicule` (ce qui est vrai pour tout objet de classe `Vehicule` et ses sous-classes, comme `Velo` et `Voiture`). La fonction `isa` du package `UNIVERSAL` va nous permettre de faire cela :

```
use UNIVERSAL qw(isa);
if( isa( $r, "Vehicule" ) ) {
   $r->roule( 40 );
}
```

On teste ainsi si la variable `$r` *est un* objet de classe `Vehicule`.

11.13 Champs et méthodes statiques

Un champ statique est un champ qui est commun à tous les objets d'une classe, c'est-à-dire qu'il n'est pas lié à une instance particulière, mais à une classe. C'est une sorte de variable globale (dont l'accès peut éventuellement être contrôlé) qui est située dans une classe.

Pour faire cela en Perl, nous utiliserons des variables déclarées dans le package de la classe en question.

```
package Vehicule;
my  $privateVar = 0;
our $publicVar = "hello";
```


Avec le qualificateur `my`, nous déclarons des variables privées (car visibles uniquement depuis l'intérieur du package). Avec `our`, sont déclarées des variables publiques (accessibles depuis n'importe quel package).

Je rappelle que pour accéder à la variable `$publicVar` du package `Vehicule`, on doit écrire `$Vehicule::publicVar`; depuis les fonctions et méthodes de ce package, il est suffisant d'écrire `$publicVar` (sauf masquage par une variable locale).

On pourrait par exemple compter le nombre de véhicules créés au moyen d'une telle variable :

```
package Vehicule;
my  $nbVehicules = 0;
sub new {
   my ($class,$nbRoues,$couleur) = @_;
   my $this = {};
   bless($this, $class);
   $this->{NB_ROUES} = $nbRoues;
   $this->{COULEUR} = $couleur;
   $nbVehicules++; # un véhicule de plus
   return $this;
}
```

À chaque appel au constructeur de cette classe, la variable `$nbVehicules` sera donc incrémentée.

Maintenant, comment écrire une méthode statique ? Une méthode statique (ou méthode de classe) est une méthode qui n'est pas appelée sur une instance de la classe (donc pas de variable `$this`), mais pour toute la classe. Ce n'est ni plus ni moins qu'une brave fonction présente dans le package. Cette fonction pourra donc uniquement accéder aux champs statiques de la classe.

Nous pourrions, par exemple, écrire une méthode statique qui renvoie le nombre de véhicules créés (variable `$nbVehicules`) :

```
sub getNbVehicules {
   my ($class) = @_;
   return $nbVehicules;
}
```

On notera que la méthode prend en premier argument le nom de la classe. Cela a pour conséquence que l'appel à la méthode ne se fait pas tout à fait comme pour une fonction d'un package (comme vu pour les modules), mais de la manière suivante :

```
print Vehicule->getNbVehicules()."\n";
```

Le nom de la classe est suivi d'une flèche, du nom de la méthode et des éventuels arguments entre parenthèses. N'écrivez pas `Vehicule::getNbVehicules()`, car le nom de la classe n'est pas transmis et surtout car les mécanismes d'héritage ne sont pas mis en œuvre. S'il est possible d'écrire `Velo->getNbVehicules()`, il n'est pas permis d'écrire `Velo::getNbVehicules()`.

Le lecteur notera que les constructeurs sont des méthodes statiques. Ils retournent des références bénies, mais n'ont rien de particulier par rapport à d'autres méthodes de classe.

Il est tout à fait possible d'appeler cette méthode sur une instance de la classe `Vehicule`

```
print $v->getNbVehicules()."\n";
```

mais dans ce cas le premier argument reçu n'est pas le nom de la classe mais l'objet en question (c'est donc une méthode d'instance et de classe...). Cela ne change rien pour notre méthode `getNbVehicules` car elle n'utilise pas son premier argument, mais le cas est gênant pour les constructeurs qui ont à bénir une référence. Pour cela, tout constructeur devrait commencer par déterminer s'il a en premier argument le nom de la classe ou une référence. L'instruction qui suit place dans la variable `$class` la classe actuelle, que cette variable ait pour valeur initiale le nom de la classe ou une instance de la classe :

```
$class = ref($class) || $class;
```

Il convient dorénavant d'écrire le constructeur ainsi :

```
sub new {
   my ($class,$nbRoues,$couleur) = @_;
   $class = ref($class) || $class;
   my $this = {};
   bless($this, $class);
   $this->{NB_ROUES} = $nbRoues;
   $this->{COULEUR} = $couleur;
   $nbVehicules++; # un véhicule de plus
   return $this;
}
```

Il est maintenant possible d'écrire la ligne suivante dans le script :

```
$v2 = $v->new( 1, "noir" );
```

Le constructeur est appelé sur une instance de la classe plutôt que sur la classe. On pourrait faire de même pour toutes les méthodes statiques (c'est même plutôt conseillé).

11.14 Exemple complet

Voici le contenu exact des fichiers d'exemple bâtis tout au long de cette partie du document.
Fichier `Vehicule.pm` :

```
package Vehicule;
use strict;
use warnings;
my  $nbVehicules = 0;
sub new {
   my ($class,$nbRoues,$couleur) = @_;
   $class = ref($class) || $class;
   my $this = {};
   bless($this, $class);
   $this->{NB_ROUES} = $nbRoues;
```



```
   $this->{COULEUR} = $couleur;
   $nbVehicules++; # un véhicule de plus
   return $this;
}
sub roule {
   my ($this,$vitesse) = @_;
   print "Avec $this->{NB_ROUES} roues, je roule à $vitesse.\n";
}
sub toString {
   my ($this) = @_;
   return "(Vehicule:$this->{NB_ROUES},$this->{COULEUR})";
}
sub getNbVehicules {
   my ($class) = @_;
   $class = ref($class) || $class;
   return $nbVehicules;
}
sub DESTROY {
   my ($this) = @_;
   print "Bye bye Vehicule ! ";
   print "($this->{NB_ROUES} $this->{COULEUR})\n";
}
1;  # À ne pas oublier...
```

Fichier `Velo.pm` :

```
package Velo;
use strict;
use warnings;
use Vehicule;
our @ISA = qw(Vehicule);
sub new {
   my ($class,$couleur,$nbVitesses) = @_;
   $class = ref($class) || $class;
   my $this = $class->SUPER::new( 2, $couleur );
   $this->{NB_VITESSES} = $nbVitesses;
   return bless($this,$class);
}
sub pedale {
   my ($this,$ici) = @_;
   print "Sur mon vélo $this->{COULEUR} ";
   print "je pédale avec $this->{NB_VITESSES} vitesses";
   print " dans $ici.\n";
}
sub toString {
   my ($this) = @_;
   my $s = "[Velo:$this->{NB_VITESSES}";
```

```
   $s .= $this->SUPER::toString();
   $s .= "]";
}
sub DESTROY {
   my ($this) = @_;
   $this->SUPER::DESTROY();
   print "Bye bye Velo ! ";
   print "($this->{NB_VITESSES} $this->{NB_ROUES} ".
          $this->{COULEUR})\n";
}
1;
```

Fichier `Garage.pm` :

```
package Garage;
use strict;
use warnings;
sub new {
   my ($class,$places) = @_;
   $class = ref($class) || $class;
   my $this = {};
   bless($this, $class);
   $this->{PLACES} = $places;
   $this->{VEHICULE} = [];
   return $this;
}
sub ajoute {
   my ($this,$vehicule) = @_;
   if( @{$this->{VEHICULE}} < $this->{PLACES} ) {
      push @{$this->{VEHICULE}}, $vehicule;
      return 1;
   }
   return 0;
}
sub toString {
   my ($this) = @_;
   my $s = "{Garage:$this->{PLACES},";
   $s .= join( ',', map( {$_->toString()}
      sort( {$a->{NB_ROUES} <=> $b->{NB_ROUES} }
            @{$this->{VEHICULE}} ) ) );
   return $s."}";
}
1;
```

Fichier `script.pl` :

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



```
use strict;
use warnings;
use Data::Dumper;
use UNIVERSAL qw(isa);
use Vehicule;
use Garage;
use Velo;
my $v = Vehicule->new( 2, "bleu" );
my $v2 = Vehicule->new( 4, "rouge" );
print "$v\n";
print Dumper($v)."\n";
$v->roule(30);
$v2 = undef;
if( 1 ) {
   my $v3 = Vehicule->new(3,'jaune');
}
foreach my $k (keys %$v) {
   print "$k : $v->{$k}\n";
}
print $v->toString()."\n";

my $g = Garage->new(3);
$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");
print $g->toString()."\n";

my @tab = (
             Velo->new('rose',15),
             Vehicule->new(3,'gris'),
             "hello",
             Velo->new('vert',21),
          );
foreach my $r (@tab) {
   if( isa( $r, "Vehicule" ) ) {
      $r->roule( 40 );
   }
}

my $velo = Velo->new('blanc',18);
$velo->pedale('les bois');
$velo->roule(10);
print $velo->toString()."\n";
```

```
print ref($velo)."\n";
print "Ouf, tout va bien !\n"
   if( ref($velo) eq "Velo" );
print Vehicule->getNbVehicules()."\n";
print Velo->getNbVehicules()."\n";
print $v->getNbVehicules()."\n";
$v2 = $v->new( 1, "noir" );
```


Conclusion

Nous sommes ici au terme de ce cours introductif à la programmation en Perl. Vous avez maintenant en main la plupart des concepts et des notions pour être autonome en Perl.

N'oubliez pas que la documentation de Perl est très bien faite et est disponible sur votre ordinateur au moyen de la commande `perldoc` : `perldoc perl` vous donne accès à la liste des thèmes consultables avec `perldoc`. Par exemple `perldoc perldata` vous explique les structures de données en Perl. Pour une fonction particulière, utilisez l'option `-f` : `perldoc -f chomp` Pour un module particulier, utilisez `perldoc` sans option : `perldoc Data::Dumper`

Vous avez maintenant en main beaucoup de notions importantes de Perl. À vous de les mettre en œuvre pour vos propres problèmes. Il n'est pas forcément facile d'exprimer son besoin en termes directement exploitables dans un langage particulier, mais en ayant un peu d'expérience de Perl, vous trouverez vite *votre* manière de résoudre un problème. Vous allez alors commencer à faire connaissance avec Tim Towtdi (There is more than one way to do it). `;-)`

L'auteur

Sylvain Lhullier `http://sylvain.lhullier.org/`

Spécialiste du langage Perl, Sylvain Lhullier propose des formations professionnelles dans le domaine de la programmation en Perl à destination des administrateurs système comme des développeurs (`http://formation-perl.fr/`), contactez-le pour tout besoin de formation Perl en présentiel. Depuis 2000, il enseigne ce langage en universités et écoles d'ingénieurs. Il est également l'auteur de conférences et d'ateliers sur la prise en main et l'utilisation de Perl.

Membre de l'association de promotion du langage Perl en France *"Les Mongueurs de Perl"* (`http://mongueurs.net/`) depuis 2002, il a été le coordinateur du groupe de travail ≪Articles≫ qui a pour vocation d'aider à la publication d'articles sur Perl et les technologies connexes. Entre 2002 et 2004, il publie, avec l'aide de ce groupe de travail, une série de sept articles dans Linux Magazine France sur le langage Perl. L'auteur remercie les autres membres de l'association pour les corrections effectuées.