



Apprenez à programmer en Perl !

Par dimitry ,
Kharec
et nohar



www.siteduzero.com

*Licence Creative Commons 7 2.0
Dernière mise à jour le 2/08/2012*

Sommaire

Sommaire	2
Lire aussi	1
Apprenez à programmer en Perl !	3
Partie 1 : Introduction à Perl	5
Qu'est-ce que Perl ?	5
Présentation	5
Quels outils faut-il pour programmer en Perl ?	7
Sous Windows	7
Sous GNU/Linux, Mac OS X, et autres UNIX	8
Exécuter un script	9
Création du programme	9
Premiers pas	11
Hello, world!	12
Premiers éléments de langage	12
À la découverte des variables	14
Le mot-clé "my"	15
Affecter une valeur à une variable	15
Les scalaires	16
Notion de typage	16
Les nombres	17
Les chaînes de caractères	18
Opérations sur les chaînes de caractères	20
Do what I mean!	21
Place à la pratique !	22
Diamonds are a girl's best friend	22
À vos claviers !	23
Les branchements conditionnels	26
Prédicats et booléens	26
Posons le contexte	26
Les booléens	27
Opérateurs de comparaison	27
Opérateurs logiques	29
La fonction defined	33
if, else, et unless	34
Où en sommes nous ?	34
if et else	35
Branchements en série	36
unless	38
Travailler avec des blocs	38
Mise en forme et indentation du code	38
Notion de portée lexicale	40
La syntaxe « en ligne »	41
TP : jeu de devinettes (première version)	42
Processus itératifs et boucles	45
La boucle de base : while	45
Ordre d'exécution	46
Opérateurs d'incréméntation et de décrémentation	47
TP : jeu du plus ou moins	48
Tirer un nombre au hasard	48
La boucle principale	49
Programme final	49
Traitement de fichiers	50
open et close	50
Lecture et écriture	51
Les fichiers et la boucle while	52
Exercices	53
La variable implicite \$_	55
La vérité sur l'opérateur <>	56
D'autres types de boucles	56
La boucle until	56
La boucle do ... while	57
La boucle for	58
La syntaxe en ligne	58



Apprenez à programmer en Perl !



Le tutoriel que vous êtes en train de lire est en **bêta-test**. Son auteur souhaite que vous lui fassiez part de vos [commentaires](#) pour l'aider à l'améliorer avant sa publication officielle. Notez que le contenu n'a pas été validé par l'équipe éditoriale du Site du Zéro.



Par

nohar et

Kharec et

dimitry

Mise à jour : 02/08/2012

Difficulté : Facile



2 162 visites depuis 7 jours, classé 68/797

Connaissez-vous Perl ? « Seulement de nom » ? Cela ne m'étonne pas !

Assez méconnu de nos jours, notamment depuis la naissance et l'envol de PHP dans les années 1990, Perl est un langage de programmation à la réputation ésotérique. Certains poussent même le vice jusqu'à le qualifier de difficile à comprendre, lui donnant l'image d'un outil terrifiant au moyen duquel les « nerds » du côté obscur de la Force produisent des programmes cryptiques que bien peu de courageux aventuriers ne tentent de modifier après leur passage. Programmer en Perl relèverait donc, dans l'imaginaire collectif, tant de l'exploit que de la magie noire...

Eh bien c'est faux !

Si vous lisez ces quelques mots, c'est que vous vous apprêtez à **apprendre à programmer en Perl**. Vous êtes donc sur le point de découvrir un langage qui a justement été conçu pour être plus **facile et convivial** à utiliser que les langages qui tiennent le devant de la scène tels que C ou Java, tout en restant un outil particulièrement puissant, capable de traiter d'énormes volumes de données en très peu de lignes de code, à tel point que ses caractéristiques ont été imitées dans la conception d'autres langages très utilisés dans le domaine du web, comme PHP ou Ruby. Il est le compagnon de route de nombreux administrateurs système sous Unix, mais aussi la matière première de plusieurs gros sites et applications connues, tels qu'[Amazon](#), [IMDb](#), [slashdot](#), les serveurs [Bugzilla](#), ou encore une partie du gestionnaire de version [git](#).

Perl est un langage pragmatique, de la puissance duquel il est facile de tirer profit afin de créer des scripts et des programmes en très peu de temps. Ces caractéristiques lui valent son surnom de *rouleau de scotch de l'Internet*. Cela ne vous est-il jamais arrivé de vous sentir démuni parce que vous n'aviez pas de scotch sous la main pour fabriquer ou réparer un objet indispensable en deux temps, trois mouvements ? 😊



À qui est destiné ce tutoriel ?

- Vous n'avez jamais codé de votre vie, mais vous souhaitez découvrir la programmation pour développer vos propres outils ou vos premiers petits jeux ?
- Vous êtes en stage ou travaillez dans une entreprise dans laquelle on vous a demandé d'écrire ou maintenir un script en Perl ?
- Vous connaissez déjà un autre langage comme C++ ou Java, mais ressentez le besoin d'un outil de beaucoup plus haut niveau pour faire communiquer vos programmes entre eux ?
- Ou bien vous êtes simplement curieux et joueur, et vous voulez passer gratuitement pour un mage noir aux yeux de vos amis geeks ?

Si vous avez répondu « oui » à l'une de ces questions, **ce cours est fait pour vous !**

En lisant ce tutoriel, qui démarre depuis zéro, vous apprendrez :

- les bases de la programmation impérative,
- les caractéristiques qui font de Perl un indispensable couteau suisse,
- comment passer d'un problème concret à l'écriture du programme qui le résoud,
- les problématiques courantes en programmation : la lecture et l'écriture dans des fichiers et dans des flux de données, comment faire communiquer des programmes sur un réseau...
- les notions de base de la programmation orientée objet.

Nous serons amenés à réaliser plusieurs programmes en suivant ce cours, tels que de petits jeux, des outils d'administration, et même un petit serveur web ! Alors, tentés ?

Partie 1 : Introduction à Perl

Dans cette première partie vous apprendrez les bases de Perl de manière à être aptes, à réaliser vos premiers scripts.

Qu'est-ce que Perl ?

Bienvenue dans le premier chapitre de ce tutoriel !

Vous apprendrez d'abord ce qu'est Perl, puis quels outils il vous faut pour programmer en Perl.

A la fin, vous exécuterez votre premier programme. 😊

Présentation

Perl est un langage de programmation créé en 1987 par Larry Wall. À l'origine, il était surtout utilisé pour le développement de scripts d'administration système sous **UNIX**, mais, avec les années, et après plusieurs révisions importantes du langage, Perl est rapidement devenu un outil polyvalent, puissant et extrêmement pratique, ce qui lui vaut aujourd'hui le surnom humoristique de « rouleau de scotch de l'internet ».

Un « langage de script »

On entend souvent parler de « *langage de script* » lorsque l'on désigne certains langages comme Perl, PHP, Python ou encore Ruby, mais que signifie cette expression, au juste ? Pour comprendre cette distinction, il nous faut examiner brièvement les différentes façons dont les programmes peuvent être exécutés sur un ordinateur.

Au départ, un programme se présente toujours comme un ensemble de fichiers qui ne contiennent **rien d'autre que du texte**. Cet ensemble de fichiers forme ce que l'on appelle le code-source du programme. Le code-source est rédigé dans un **langage de programmation** qui est compréhensible par les humains (du moins ceux qui savent programmer 😊). Il existe de très nombreux langages de programmation, dont Perl fait partie.



Larry Wall, créateur de Perl

Crédit photo : Randal Schwartz

Lorsque l'on veut exécuter un programme dont on possède le code-source, la première chose à faire est de le **compiler**. La compilation, c'est simplement le fait de **transformer** un programme écrit dans le langage X afin de lui donner une *représentation abstraite* que l'ordinateur peut comprendre. Typiquement, cette représentation abstraite modélise une série d'instructions. Cette étape de compilation est réalisée par ce que l'on appellera pour l'instant un *super-programme*.

Après la phase de compilation, il peut se passer plusieurs choses, selon le langage (ou plutôt la technologie) utilisé(e) :

- Le *super-programme* peut décider d'exécuter directement les instructions qu'il vient de compiler. Dans ce cas de figure, le *super-programme* s'appelle **un interpréteur**.
- Alternativement, le *super-programme* peut décider de créer un nouveau fichier, dans lequel il va écrire les instructions qu'il vient de compiler, dans un langage compréhensible cette fois directement par l'ordinateur (le langage machine), et ce *super-programme* s'appelle alors, simplement, et par abus de langage, **un compilateur**.

Il existe encore d'autres cas de figure possibles que nous taïrons ici (ce n'est pas le sujet), mais ce qu'il faut retenir, c'est simplement que dans un cas, le programme que l'on vient de compiler est exécuté par l'interpréteur, alors que dans l'autre cas, le programme peut être exécuté directement par la machine. Dans ces conditions, ce que l'on appelle un langage de script, c'est un langage avec lequel les programmes sont compilés et interprétés dans la foulée. Ceci a l'avantage de rendre les programmes faciles à maintenir et à corriger (sans passer par une étape de compilation indépendante qui peut parfois être longue).

En ce qui nous concerne, les programmes (ou scripts) que l'on écrit en Perl sont exécutés par un interpréteur qui se nomme... perl.



À partir de maintenant, nous ferons toujours référence au langage en écrivant Perl avec un P majuscule, et à l'interpréteur en écrivant perl avec un p minuscule.

D'autres, parmi lesquels le créateur de Perl, Larry Wall, considèrent que ce qui définit les langages de scripts est leur haut niveau d'abstraction (c'est-à-dire qu'ils évitent au développeur de gérer manuellement des aspects rébarbatifs de la programmation, tels que la gestion de la mémoire), et dont le but avoué est d'être simple à utiliser afin que « *ce qui est possible devienne facile* », et que « *ce qui est difficile devienne possible* ».

Que peut-on programmer en Perl ?

Comme on l'a dit plus haut, bien que Perl ait initialement été conçu pour être un langage de scripts d'administration système sous UNIX (rôle qu'il remplit encore très bien aujourd'hui), il s'agit véritablement d'un langage généraliste (y compris sous Windows), et qui est employé sur des projets de toutes tailles, soit comme langage principal, soit, comme en atteste son surnom, à la façon d'un « rouleau de scotch » grâce auquel on peut faire communiquer des programmes entre eux pour former de gros systèmes.

Évidemment, là où il est le plus fort et le plus intéressant, cela reste le traitement de fichiers et de données, mais grâce à ses nombreux modules, Perl permet de produire très rapidement des programmes capables de se connecter à Internet, des sites et des applications web ou des logiciels avec des interfaces graphiques ! À titre d'exemples d'applications et de sites web connus qui utilisent Perl aujourd'hui, on pourra citer *BugZilla*, *Amazon.com* ou encore *IMDb*.

La philosophie de Perl

Tous les langages de programmation s'articulent autour d'une philosophie de base qui les définit. Celle de Perl tient globalement en une poignée de devises :

- « *Do what I mean* ».
Perl est un langage réputé pour accomplir la volonté du programmeur plutôt que d'exécuter "bêtement" ce qu'on lui dit. Cela signifie que le langage est assez intelligent pour comprendre, suivant le contexte, ce que l'on veut faire. Ainsi, pour faire accomplir une tâche donnée à Perl, on peut la plupart du temps se contenter de formuler les étapes principales de l'algorithme : le langage est conçu de manière à exécuter automatiquement la plupart des conversions et des micro-tâches que requiert chaque étape, que l'on serait obligé d'explicitier dans un langage de moins haut niveau d'abstraction tel que C ou C++.
- « *There is more than one way to do it* ».
En Perl, pour réaliser une opération donnée, ou résoudre un problème, il existe toujours une multitude de solutions différentes, plus ou moins évidentes, plus ou moins explicites, plus ou moins élégantes, plus ou moins efficaces. Cela signifie que lorsque l'on cherche à coder rapidement quelque-chose et que l'on est habitué à utiliser Perl, on se retrouve rarement sans ressource, à devoir réfléchir pendant longtemps : il nous vient toujours au moins une solution directe pour résoudre le problème.
- « *Things that are different should look different* ».
Il s'agit là de l'un des aspects de Perl sur lesquels on plaisante le plus souvent : un code-source écrit en Perl a la réputation d'être *moche* à regarder. Cela est dû au choix du créateur de Perl de faire en sorte que les éléments du langage de natures ou de fonctions différentes soient facilement reconnaissables visuellement, en usant notamment de symboles (les fameux *sigils*). En usant de ces symboles, le programmeur Perl peut exprimer avec très peu de texte des instructions compliquées, et comprendre en un coup d'œil ce à quoi chaque mot fait référence.

Vous l'aurez compris : Perl est un langage orienté pratique, qui est conçu pour *accomplir le boulot*, quel qu'il soit, et quels que soient les moyens employés. C'est un langage pragmatique, et c'est la raison pour laquelle il est autant affectionné par les administrateurs systèmes qui ont souvent besoin de réaliser un script en 3 minutes montre en main pour accomplir et automatiser une tâche donnée. 😊

Enfin, on pourra noter que la communauté Perl n'est pas dénuée d'humour. Le fait qu'il existe toujours plusieurs façons d'exprimer la même chose en Perl permet aux développeurs qui pratiquent ce langage de réaliser, pour s'amuser, les scripts les plus courts, ou les moins lisibles dont ils soient capables (notamment des *one-liners*, ces programmes complets qui tiennent en une ligne) : il s'agit d'une activité amusante et stimulante, qui demande souvent de réfléchir à la manière d'exprimer une instruction de la façon la plus concise possible.

Perl 5 et Perl 6



Logo de Perl 5 (à gauche) et de Perl 6 (à droite)

Actuellement, il existe deux versions indépendantes de Perl, appelées trompeusement Perl 5 et Perl 6.

La version que l'on appelle « Perl 5 » est la cinquième version du Perl d'origine, dont le développement est dirigé par Larry Wall en personne. Cette version de Perl existe et évolue depuis 1994 : c'est le Perl d'aujourd'hui, la version qui est installée par défaut sur tous les systèmes d'exploitation dérivés d'UNIX, et qui est de très loin la plus utilisée dans le monde.

Parallèlement, depuis quelques années, un « nouveau » Perl est en cours de développement, baptisé Perl 6. Il s'agit d'un langage différent qui partage seulement la philosophie de base et la syntaxe de Perl 5. Perl 6, à l'heure où sont écrites ces lignes, est encore en cours de développement, à un stade expérimental. Cela signifie qu'il faudra encore attendre un moment avant d'obtenir une version stable du langage, qui puisse être utilisée en production.

Étant donné l'aspect pragmatique du langage, ce cours est basé sur Perl 5.

Quels outils faut-il pour programmer en Perl ?

Pour travailler avec Perl, vous allez avoir besoin de deux programmes : un interpréteur Perl et un éditeur de texte.

Sous Windows

L'interpréteur Perl

Pour installer Perl sur votre machine, rendez vous sur <http://strawberryperl.com/>, puis cliquez sur la version qui correspond à votre ordinateur. Si vous ne savez pas laquelle prendre, choisissez la version "32-bit". Ensuite, contentez-vous de lancer l'installateur que vous venez de télécharger, et laissez-vous guider par les instructions qui s'affichent à l'écran.

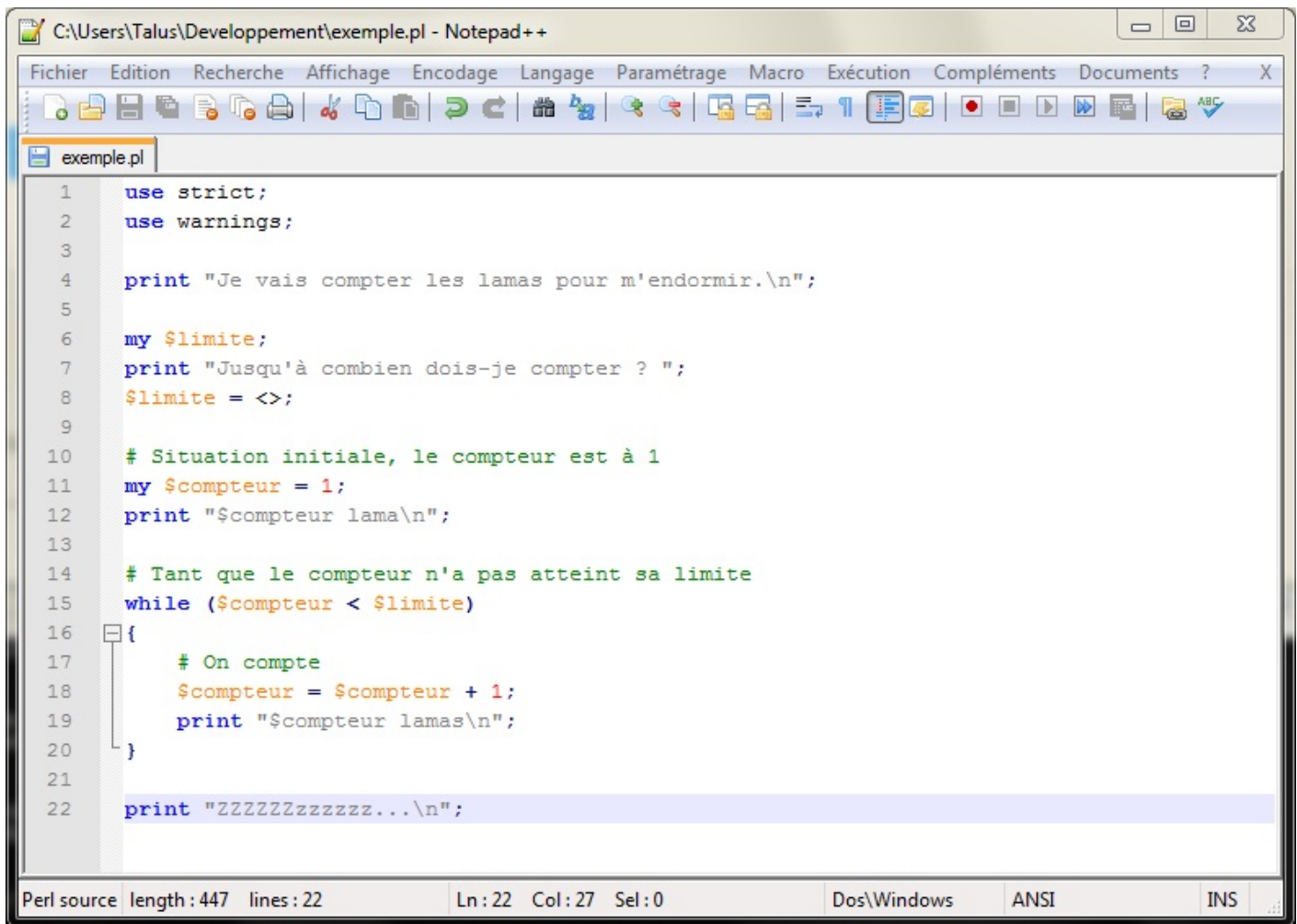
Perl est maintenant installé sur votre machine.

Si ce n'est pas le cas, recherchez sur Google l'erreur qui apparaît à l'écran : il s'agit bien souvent de la méthode la plus rapide pour solutionner un problème. Si, malgré tout, vous ne trouvez toujours pas, vous pouvez demander de l'aide sur les forums du Site du Zéro.

L'éditeur de texte

Sous Windows, vous avez le fameux « Bloc-notes » installé par défaut. Il est assez simple d'utilisation. Le seul inconvénient est qu'il ne gère pas la coloration syntaxique, c'est-à-dire le système qui permet d'afficher du code en coloriant les mots selon leur signification, et ainsi d'avoir une lecture plus aisée du programme.

Pour pallier ce problème, il existe un éditeur de texte très pratique, nommé Notepad++ (n++ pour les intimes) qui supporte de nombreuses options utiles pour les programmeurs, dont la coloration syntaxique.



The screenshot shows a Notepad++ window titled 'C:\Users\Talus\Developpement\exemple.pl - Notepad++'. The menu bar includes 'Fichier', 'Edition', 'Recherche', 'Affichage', 'Encodage', 'Langage', 'Paramétrage', 'Macro', 'Exécution', 'Compléments', 'Documents', '?', and 'X'. The toolbar contains various icons for file operations, editing, and execution. The script content is as follows:

```
1 use strict;
2 use warnings;
3
4 print "Je vais compter les lamas pour m'endormir.\n";
5
6 my $limite;
7 print "Jusqu'à combien dois-je compter ? ";
8 $limite = <>;
9
10 # Situation initiale, le compteur est à 1
11 my $compteur = 1;
12 print "$compteur lama\n";
13
14 # Tant que le compteur n'a pas atteint sa limite
15 while ($compteur < $limite)
16 {
17     # On compte
18     $compteur = $compteur + 1;
19     print "$compteur lamas\n";
20 }
21
22 print "ZZZZZZzzzzzz...\n";
```

The status bar at the bottom indicates 'Perl source', 'length : 447', 'lines : 22', 'Ln : 22', 'Col : 27', 'Sel : 0', 'Dos\Windows', 'ANSI', and 'INS'.

Vous le trouverez à cette adresse : <http://notepad-plus-plus.org/fr>.

Téléchargez-le, puis installez-le comme tout autre programme.

Ça y est, tout s'est bien passé ? Vous pouvez passer à la partie "Exécuter un script".

Sous GNU/Linux, Mac OS X, et autres UNIX

Sur les systèmes d'exploitation compatibles POSIX, vous n'allez pas avoir besoin de chercher bien loin pour installer Perl : il est déjà sur votre machine !

Pour vous en convaincre, ouvrez un terminal puis tapez la commande suivante :

Code : Console

```
perl -v
```

Perl vous répondra alors un petit texte du même goût que ceci :

Code : Console

```
This is perl 5, version 14, subversion 2 (v5.14.2) built for i686-linux-
thread-multi
```

```
Copyright 1987-2011, Larry Wall
```


Perl may be copied only under the terms of either the Artistic License or the GNU General Public License, which may be found in the Perl 5 source kit.

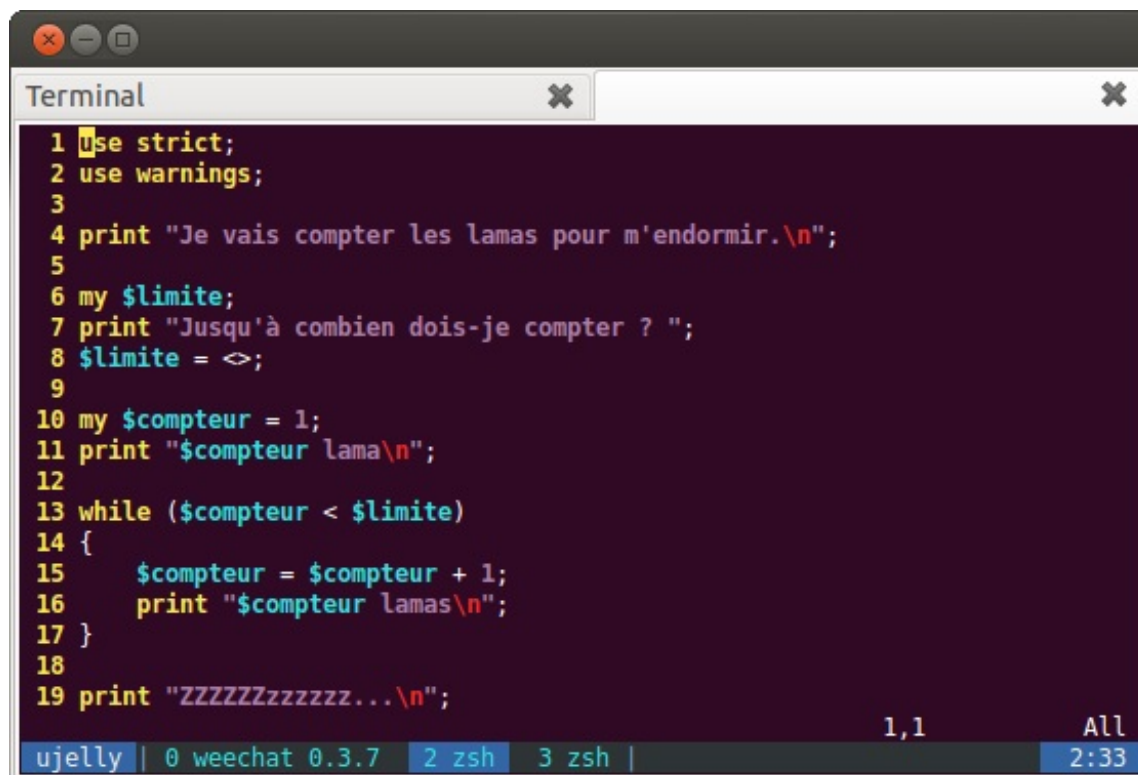
Complete documentation for Perl, including FAQ lists, should be found on this system using "man perl" or "perldoc perl". If you have access to the Internet, point your browser at <http://www.perl.org/>, the Perl Home Page.

Il est possible que votre version de Perl diffère de celle qui est affichée ici. Si c'est le cas, ne paniquez pas ! Du moment qu'il s'agit de Perl 5.x, vous pourrez largement suivre l'intégralité de ce cours sans que cela ne pose le moindre problème. 😊

Choix d'un éditeur de texte

Sous GNU/Linux, en ce qui concerne les éditeurs de texte ou les environnements de développement pour travailler avec Perl, vous avez l'embarras du choix, tous les éditeurs gèrent la coloration syntaxique !

Citons, à titre d'exemple gEdit et Geany, qui sont plutôt adaptés aux environnements Gnome et Unity, Kate, qui est, lui, plutôt fait pour KDE, ou encore, les incontournables et inénarrables éditeurs stars : Vim et Emacs, qui demanderont aux débutants un temps certain d'adaptation ainsi que de faire un choix cornélien (il faut choisir l'un ou l'autre), mais s'avéreront être, sur le long terme, des outils de développement complets, puissants, et irremplaçables (ainsi qu'une source de troll intarissable).



```
1 use strict;
2 use warnings;
3
4 print "Je vais compter les lamas pour m'endormir.\n";
5
6 my $limite;
7 print "Jusqu'à combien dois-je compter ? ";
8 $limite = <>;
9
10 my $compteur = 1;
11 print "$compteur lama\n";
12
13 while ($compteur < $limite)
14 {
15     $compteur = $compteur + 1;
16     print "$compteur lamas\n";
17 }
18
19 print "ZZZZZZzzzzzz...\n";
```

Vim sous Ubuntu

Exécuter un script

Maintenant que vous avez tous les outils nécessaires pour coder en Perl, nous allons enfin réaliser notre premier programme !



Création du programme

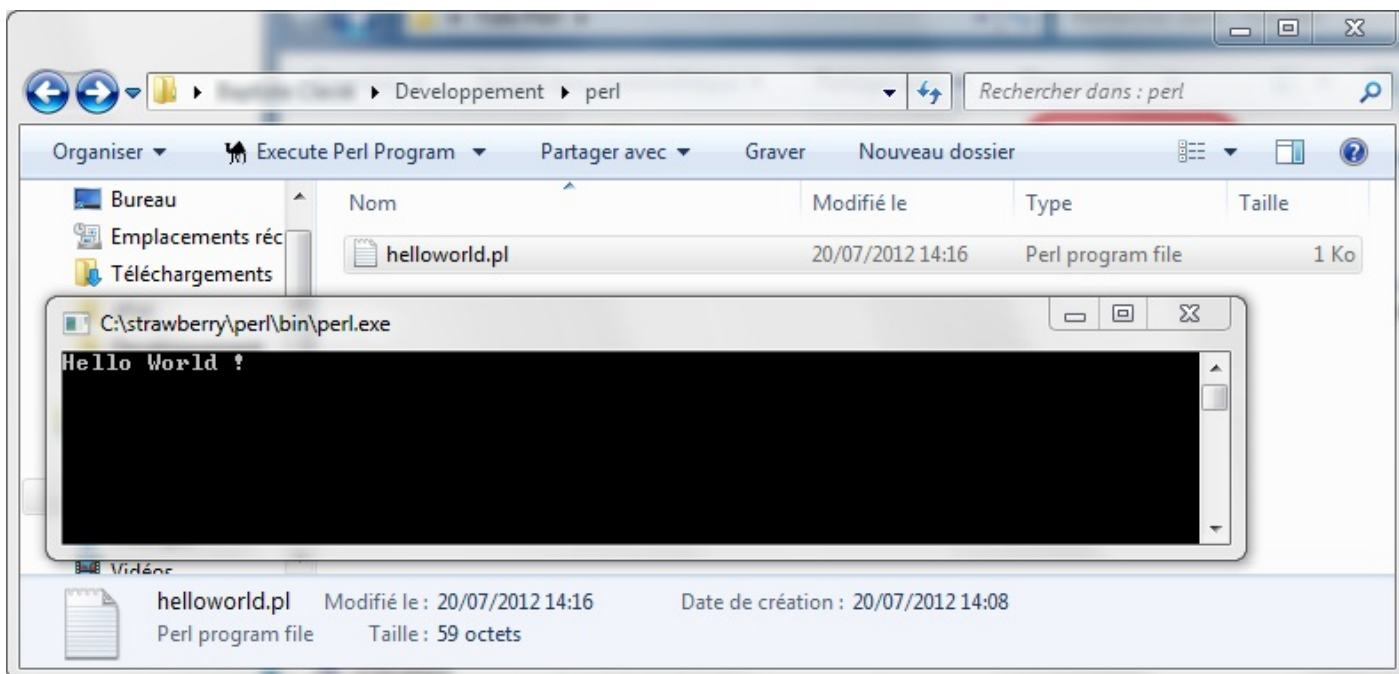
Ouvrez votre éditeur préféré, puis créez un nouveau programme et enregistrez-le sous le nom de "helloworld.pl".

Code : Perl

```
use strict;
use warnings;
```

```
print "Hello, world!";  
<>;
```

Il vous suffira, sous Windows, ensuite de double-cliquer sur l'icône du script afin de lancer celui-ci, ce qui devrait vous donner un résultat similaire à ceci :



Facile, non ?

Lancer le programme en ligne de commande

Si vous travaillez sous UNIX, il y a des chances pour que vous fassiez partie de ces gens qui n'ont pas peur d'une simple ligne de commande. 😊

Pour ouvrir une console sous Windows, appuyez simultanément sur les touches Windows et R, puis tapez "cmd" dans le champ qui apparaît.

Pour exécuter un script en ligne de commande, la façon la plus simple de faire est de lancer explicitement perl dans la console, comme ceci :

Code : Console

```
perl /chemin/vers/votre/script/helloworld.pl
```

Sous Linux et UNIX uniquement : le shebang

Alternativement, sous les systèmes compatibles UNIX, vous pouvez rajouter cette ligne (que l'on appelle un **shebang**) au tout début de votre script (obligatoirement en première ligne) :

Code : Perl

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

Cette ligne vous permet d'exécuter votre script sans avoir besoin d'appeler explicitement l'interpréteur. Pour cela, il faut que vous vous donniez les droits d'exécution sur votre script, comme ceci :

Code : Console

```
cd /dossier/du/script
chmod +x helloworld.pl      # Active les droits d'exécution
./helloworld.pl             # Lance le script
```



Pouah, mais c'est moche !

Il faudra vous y habituer, parce que nous travaillerons pendant très longtemps en console comme ici. L'avantage de la console est qu'elle permet d'apprendre les rudiments du langage sans nous pencher sur le côté graphique qui n'est pas encore à votre portée.

Vous savez maintenant à qui ressemble Perl et comment vous en servir. Dans le prochain chapitre vous ferez vos tous premiers pas avec ce langage !

Premiers pas

Il est temps maintenant de nous lancer dans l'apprentissage de Perl, et de créer nos premiers programmes.

Dans ce chapitre, nous allons apprendre beaucoup de choses fondamentales ! En particulier nous allons découvrir :

- Ce qu'est une fonction, un argument, une variable... ;
- La façon dont Perl représente les données ;
- Une manière de récupérer la saisie de l'utilisateur au clavier ;
- Ce qui différencie Perl de la plupart des autres langages de programmation.

Soyez donc très attentifs, car c'est ici que tout va se jouer. 😊

Hello, world!

Depuis toujours, la coutume veut que lorsque l'on apprend un nouveau langage de programmation, le premier programme que l'on écrit ne fasse qu'une chose très simple : dire bonjour au nouveau monde qui s'ouvre à nous. Nous n'allons bien évidemment pas déroger à la règle. 😊

Ouvrez votre éditeur de texte, et tapez le programme suivant :

Code : Perl

```
use strict;
use warnings;

print "Hello, world!";
<>;
```

Sauvegardez ce fichier en lui donnant le nom `helloworld.pl`.

Lorsque vous exécutez ce programme, le texte `Hello, world!` devrait apparaître dans la console. Appuyez sur la touche Entrée ↵ pour quitter le programme.

Premiers éléments de langage

Le programme que nous venons d'écrire comporte quatre **instructions** qui se terminent chacune par un point-virgule et que perl a exécutées l'une après l'autre.

Les deux premières instructions que nous avons écrites sont `use strict;` et `use warnings;`. Pour dire les choses simplement, elles servent à rendre Perl moins permissif, ce qui nous sera extrêmement utile pour détecter rapidement de nombreuses erreurs dans le code (comme, par exemple, le fait d'oublier le point virgule à la fin d'une instruction). Ces deux instructions figureront en préambule de tous les programmes que nous écrirons dans ce cours.

La troisième instruction que nous avons écrite est de loin la plus intéressante : il s'agit du cœur du programme lui-même.

Code : Perl

```
print "Hello, world!";
```

Une instruction est un ordre

Tout d'abord, il nous faut remarquer que chaque instruction que vous écrirez en Perl est un ordre qui est donné à la machine. En effet, les trois instructions que nous avons écrites jusqu'ici sont construites de la même façon qu'une phrase décrivant un ordre dans un langage naturel comme le français ou l'anglais :

VERBE COMPLÉMENTS,

où le VERBE est conjugué à l'impératif, comme dans l'injonction « fais tes devoirs ! ». En Perl, nous n'allons pas parler d'un VERBE, mais plutôt d'une **fonction**. Dans notre exemple, il s'agit de la fonction **print**, qui signifie *afficher* en anglais, et sert ici à écrire du texte à l'écran. De la même façon qu'un VERBE accepte un ou plusieurs COMPLÉMENTS, on dit que les fonctions en Perl acceptent un ou plusieurs **arguments**. Ici, l'argument que nous avons passé à la fonction **print** est le texte `"Hello, world!"`.

Remarquez qu'en Perl, les instructions suivantes sont strictement équivalentes :

Code : Perl

```
print "Hello, world!";
```

Code : Perl

```
print      "Hello, world!"      ;
```

Code : Perl

```
print("Hello, world!");
```

Code : Perl

```
print  
"Hello, world!"  
;
```

Comme vous le voyez, Perl est très tolérant sur la façon dont il lit les instructions que vous lui donnez. Néanmoins, dans ce cours, nous allons faire l'effort de toujours présenter nos programmes de façon claire et cohérente, pour les rendre le plus simple possible à lire et à corriger. C'est une habitude vitale, lorsque l'on code !

Mettre le programme en pause

Tout à la fin de notre script, nous avons rajouté une dernière ligne un petit peu particulière :

Code : Perl

```
<>;
```

Cette ligne, qui n'est pas indispensable au fonctionnement du programme, sert simplement ici à marquer une pause dans l'exécution de votre script, jusqu'à ce que l'utilisateur appuie sur la touche Entrée.

Ceci est très pratique sous Windows si vous décidez d'exécuter votre script en double-cliquant sur le fichier. En effet, sans elle, vous n'auriez pas le temps de lire le contenu de la console qui apparaît : celle-ci se ferme aussitôt l'exécution terminée. En revanche, si vous travaillez sous Linux, il y a des chances que cette ligne soit inutile. A fortiori si, comme l'auteur de ce chapitre, vous prenez l'habitude d'exécuter vos scripts dans la console, en appelant perl en ligne de commande, ce comportement peut même s'avérer contre-intuitif et gênant.



De façon à ne pas surcharger nos scripts dans toute la suite de ce cours, nous ne ferons pas systématiquement figurer



cette ligne dans les codes suivants. Ce sera donc à vous de l'ajouter le cas échéant, selon votre façon de travailler.

Les commentaires

Une autre bonne habitude à prendre est celle de placer des commentaires dans nos programmes, de façon à expliquer ce qui s'y passe. En Perl, les commentaires sont introduits par un dièse (#), et se poursuivent jusqu'à la fin de la ligne.

Voici par exemple notre premier programme, agrémenté de quelques commentaires :

Code : Perl

```
# Ces instructions rendent Perl moins permissif
use strict;
use warnings;

print "Hello, World!"; # Affiche "Hello, World!" dans la console
<>; # Met le programme en pause jusqu'à ce que l'utilisateur appuie
    # sur la touche "Entrée"
```

Avant de passer à la suite, vous pouvez vous amuser à modifier ce script afin qu'il affiche autre chose que "Hello, World!" dans la console, par exemple. En particulier, vous pouvez **chercher à faire planter ce script !**

En effet, c'est en découvrant par vous-mêmes tout ce qui est susceptible de déclencher des erreurs que vous retiendrez la syntaxe de Perl. 😊

À la découverte des variables

Nous allons maintenant étudier un nouveau script. Écrivez le programme suivant dans un nouveau fichier et exécutez-le :

Code : Perl

```
use strict;
use warnings;

my $nom = "Perl";
print "$nom c'est coolympique !";
```

Voici ce que cela va donner dans la console :

Code : Console

```
Perl c'est coolympique !
```

Remarquez que la ligne 4 présente bon nombre d'éléments nouveaux. Afin de mieux examiner ce qui s'y passe, décomposons cette ligne en deux instructions :

Code : Perl

```
my $nom;
$nom = "Perl";
```

Nous allons analyser ces deux lignes en détail.

Le mot-clé "my"

La première nouveauté que nous avons croisée dans ce script est l'utilisation du mot-clé **my** (le possessif « mon » en anglais). Ce mot-clé sert à **déclarer une variable**, c'est-à-dire, intuitivement, créer une boîte vide, sur laquelle on colle une étiquette pour décrire ce qu'elle va contenir. Ici, nous avons appelé notre variable `$nom`, mais nous aurions tout aussi bien pu l'appeler `$Jean_Pierre`, `$choucroute` ou `$KevinDu35`. Les variables sont simplement des « conteneurs », des récipients dans lesquels on peut mettre des objets, nous sommes donc libres de coller sur ces récipients les étiquettes que l'on veut !

Enfin presque : il y a quand même deux règles simples à suivre pour nommer nos variables.

D'abord, les noms de variables commencent tous par un symbole spécial ; ici le "dollar" : `$`. Ce symbole, que l'on appelle un « *sigil* », a une signification très précise en Perl, que nous étudierons beaucoup plus loin dans ce cours. Pour l'heure, nous allons commencer par utiliser uniquement des "variables dollar", qui portent le nom de **scalaires** (*scalar* en anglais).

La deuxième règle à respecter, c'est que les noms des variables doivent se composer uniquement de caractères alpha-numériques (c'est-à-dire de chiffres et de lettres), majuscules ou minuscules, ou d'underscores (`_`), sans espace.

En Perl, lorsque vous utilisez le pragma `strict` (avec `use strict;`), vous devez déclarer vos nouvelles variables avec le mot-clé **my** avant de les utiliser. Bien que cela semble contraignant, vous vous rendrez très vite compte qu'il s'agit d'un moyen de détecter certaines erreurs d'inattention bêtes et méchantes, comme une faute de frappe dans le nom d'une variable, par exemple.

Affecter une valeur à une variable

Pour placer une valeur dans une variable, on utilise l'opérateur `=` comme dans la ligne suivante :

Code : Perl

```
$nom = "Perl"; # J'affecte la valeur "Perl" à la variable $nom.
```

Ainsi, notre variable `$nom` contient maintenant la valeur `"Perl"`. Chaque fois que vous utiliserez cette variable `$nom` dans votre script, celle-ci sera remplacée par la valeur `"Perl"` jusqu'à ce que vous lui affectiez une nouvelle valeur.

C'est en particulier ce qu'il s'est passé à la ligne suivante :

Code : Perl

```
print "$nom c'est coolympique !";
```

Lorsqu'il a interprété cette ligne, perl a remplacé `$nom` par la valeur `"Perl"` dans la chaîne `"$nom c'est coolympique !"`, afin de former une nouvelle chaîne de caractères : `"Perl c'est coolympique !"`. On appelle ce phénomène *l'interpolation* ; nous allons y revenir dans quelques instants.

Exercice

Avant de poursuivre ce chapitre et d'étudier les scalaires en détail, essayez de deviner ce que les codes suivants vont afficher dans la console :

Code : Perl

```
my $a = "Bonjour";  
my $b = "madame";  
print "$a, $b.";
```


Code : Perl

```
my $age = 13;
print "PS : j'ai $age ans.";
```

Code : Perl

```
my $a = 3;
my $b = 2;
my $somme = $a + $b;
print "$a + $b = $somme";
```

Les scalaires

Comme nous l'avons vu plus haut, les variables qui commencent par le *sigil* `$` sont appelées en Perl des *scalaires*. Cette notion de *scalaire* est l'une de celles qui font de Perl un langage très pratique à utiliser, puisqu'elle découle directement de l'une des devises de la philosophie de Perl : « *Do what I mean* ».

Notion de typage

Le *typage* est une notion que l'on retrouve dans tous les langages de programmation. Bien qu'elle puisse s'exprimer de plein de façons différentes, elle reflète toujours cette même vérité simple et profonde : *on ne mélange pas les torchons et les serviettes !* En programmation, ce dicton signifie simplement que les données que l'on manipule ont un type, et que l'on ne peut pas faire n'importe quoi avec n'importe quelle donnée de n'importe quel type. Au travers des exemples de code que nous avons examinés jusqu'à présent, vous avez rencontré sans le savoir deux *types de données* différents : les chaînes de caractères (comme `"hello"`) et les nombres entiers (comme `42`).

Vous viendrait-il à l'esprit de calculer la somme de `"hello"` et de `42` ? Cela n'aurait aucun sens, n'est-ce pas ?

Tenez, soyons fous, et essayons quand même pour voir ce qu'en dit Perl :

Code : Perl

```
use strict;
use warnings;

my $a = "hello";
my $b = 42;
my $somme = $a + $b;
print "Resultat: $somme";
```

Voici ce qu'il nous répond :

Code : Console

```
Argument "hello" isn't numeric in addition (+) at test.pl line 6.
Resultat: 42
```

La première ligne de la réponse de perl est ce que l'on appelle un *warning* : perl vous dit grosso-modo que vous venez de lui demander de faire quelque-chose qu'il a du mal à comprendre, à savoir ajouter 42 à une valeur *non numérique*. Il vous précise d'ailleurs que c'est à la ligne 6 que cela se passe. Comme vous le voyez, perl répugne à mélanger les torchons et les serviettes.

Cela dit, vous remarquerez qu'il affiche quand même sur la deuxième ligne le résultat du programme, ce qui signifie qu'il a quand même réalisé le calcul en considérant que la valeur qu'il ne comprenait pas (`"hello"`) valait 0. Voilà qui est étrange : dans pratiquement n'importe quel autre langage de programmation, ce script aurait planté !

Cet exemple illustre ce que l'on appelle la **force** du typage. Lorsqu'un langage est fortement typé, il déclenche des erreurs chaque fois que vous tentez de réaliser une opération avec des données du mauvais type, ou de types incompatibles entre eux. À l'inverse, lorsque le typage est faible, le langage se montre plus conciliant, et essaye de vous fournir un résultat. On dira donc que Perl est un langage faiblement typé. En réalité, Perl regroupe des données de natures différentes sous un même type : les scalaires.

Nous allons ici examiner les types les plus importants : les nombres et les chaînes de caractères. Nous en découvrirons d'autres au fur et à mesure que nous avancerons dans les chapitres de ce cours.

Les nombres

Les données les plus simples que l'on puisse traiter en programmation sont probablement les nombres. En toute rigueur, on peut distinguer au moins deux types de nombres : les nombres entiers et les nombres décimaux (ou, vulgairement, « nombres à virgule »). Bien que l'interpréteur perl sache faire la différence entre les deux dans son fonctionnement interne, celle-ci n'est que très peu marquée dans le langage, comme en atteste le script suivant :

Code : Perl

```
use strict;
use warnings;

my $a = 4;
my $b = 5.5;
my $somme      = $a + $b; # addition
my $difference = $a - $b; # soustraction
my $produit    = $a * $b; # multiplication
my $quotient   = $a / $b; # division

print "$a + $b = $somme\n";
print "$a - $b = $difference\n";
print "$a * $b = $produit\n";
print "$a / $b = $quotient\n";
```

Voici le résultat :

Code : Console

```
4 + 5.5 = 9.5
4 - 5.5 = -1.5
4 * 5.5 = 22
4 / 5.5 = 0.727272727272727
```

Pour peu que vous ayez déjà utilisé une calculatrice dans votre vie, ce code ne devrait vous poser aucun problème. 😊

On pourra noter aussi l'*opérateur d'exponentiation* `**`, qui sert à exprimer une puissance :

Code : Perl

```
my $quatre_au_carre = 4 ** 2 # 16
my $racine_de_quatre = 4 ** (1/2) # 2
```

Il existe encore beaucoup (beaucoup, beaucoup...) d'autres opérateurs que l'on peut utiliser sur des nombres en Perl, mais il ne serait pas judicieux de vous les présenter dès maintenant. Nous nous contenterons pour l'instant de ceux que nous venons de voir, et gardons la découverte des autres pour plus tard. Si vous êtes curieux, vous pouvez toujours consulter [la page "perlop"](#) (en anglais) de la documentation de Perl, sur laquelle vous les trouverez tous. Gare à l'indigestion !

Ce qu'il faut retenir, c'est que Perl peut manipuler des nombres, que les nombres sont des scalaires, et que l'on n'a pas besoin de traiter différemment les nombres entiers et les nombres décimaux : la plupart du temps, on se moque complètement de la différence, et ça, Perl l'a bien compris !

Les chaînes de caractères

Les chaînes de caractères sont probablement le type le plus important en Perl. En effet, Perl est délicieusement pratique pour manipuler du texte de façon simple et intuitive. C'est d'ailleurs sans conteste LE langage de programmation le plus doué pour ça, à tel point que de nombreux autres langages, comme PHP et Ruby, s'inspirent librement de ce savoir-faire.

Plusieurs façons d'écrire une chaîne

Jusqu'ici, nous avons toujours présenté nos chaînes de caractères de la même façon, c'est-à-dire en les entourant de guillemets doubles (ou *double-quotes*), comme ceci :

Code : Perl

```
my $chaine = "Je suis une chaine de caractères";
```

Mais que faire si nous voulons afficher une chaîne qui contient des guillemets ?

Nous avons pour cela au moins deux solutions. La première, c'est d'*échapper* les guillemets au moyen d'un antislash (\), comme ceci :

Code : Perl

```
my $chaine = "\"Cessons ce marivaudage !", dit le marquis.";
print $chaine;
```

Ce qui donne en console :

Code : Console

```
"Cessons ce marivaudage !", dit le marquis.
```

La seconde solution, c'est d'utiliser une représentation alternative pour notre chaîne, en l'entourant de guillemets simples :

Code : Perl

```
my $chaine = '"Cessons ce marivaudage !", dit le marquis.';
```

Nous allons maintenant nous attarder un petit peu sur ce qu'impliquent ces deux solutions.

Les caractères échappés et les caractères spéciaux

Nous venons de voir qu'au moyen d'un antislash, il était possible d'*échapper* des guillemets. Cette pratique est très courante en programmation, et permet non seulement d'éviter des confusions, mais aussi de représenter des caractères invisibles, comme les passages à la ligne ("`\n`") ou les tabulations ("`\t`") :

Code : Python

```
print "Ceci est un passage a la ligne : \nTadaaaa!\n\n";  
print "Les\ttabulations\tsont\tpratiques\n";  
print "pour\taligner\t\tdu\ttexte.";
```

Jugez par vous-mêmes :

Code : Console

```
Ceci est un passage a la ligne :  
Tadaaaa!  
  
Les      tabulations      sont      pratiques  
pour     aligner          du       texte.
```

Mais alors, si l'antislash permet d'échapper des caractères spéciaux, comment diable pourrais-je afficher un antislash ? vous demandez-vous certainement. Eh bien, en l'échappant lui-même, tout simplement :

Code : Perl

```
print "Ceci est un antislash : \\";
```

Code : Console

```
Ceci est un antislash : \
```

Comme pour les opérateurs mathématiques, il existe de très nombreux caractères spéciaux en Perl, et, là aussi, il serait inutile de vous en dresser une liste complète que vous ne sauriez retenir. Nous nous contenterons donc, cette fois encore, de les découvrir au fur et à mesure que nous en aurons besoin dans ce cours, par la pratique.

Interpolation et guillemets simples

Depuis le début de ce chapitre, nous utilisons l'interpolation pour écrire le contenu de nos variables dans des chaînes de caractères. Il est important de noter, à ce sujet, que les chaînes délimitées par des guillemets simples ne se comportent absolument pas de la même façon que les autres. Regardez :

Code : Perl

```
my $un_fruit = "une pomme";  
print "Hier, j'ai mangé $un_fruit.\n";  
print 'Hier, j\'ai mangé $un_fruit.\n';
```

Code : Console

```
Hier, j'ai mangé une pomme.  
Hier, j'ai mangé $un_fruit.\n
```

Comme vous le constatez, dans la seconde chaîne, la variable `$un_fruit` n'a pas été remplacée par sa valeur, et le caractère

spécial "`\n`" n'a pas été remplacé par un retour à la ligne. Par contre, l'apostrophe a bien été échappée, elle.

Ainsi, le choix d'utiliser des guillemets simples ou doubles pour délimiter les chaînes de caractères en Perl n'est pas qu'une simple affaire de goût : **cela dépend avant tout du fait de vouloir que les variables et les caractères spéciaux soient interpolés ou non.**

Opérations sur les chaînes de caractères

Comme on vient de le dire, Perl permet de réaliser beaucoup d'opérations différentes sur les chaînes de caractères. Découvrons-en quelques unes ensemble. 😊

Longueur d'une chaîne

La longueur d'une chaîne de caractères, c'est tout simplement le nombre de caractères qu'elle contient. Celle-ci peut être déterminée au moyen de la fonction `length` (qui signifie... « longueur », en anglais).

Code : Perl

```
my $chaine = "coolympique";  
my $longueur = length $chaine;  
print "La chaine '$chaine' contient $longueur caracteres.";
```

Code : Console

```
La chaine 'coolympique' contient 11 caracteres.
```

Répétition et concaténation

C'est un fait établi et connu de longue date : les programmeurs sont des gens flemmards qui n'aiment pas se répéter. Aussi, ils préfèrent laisser leurs ordinateurs répéter les choses à leur place. C'est pourquoi Perl possède un opérateur nommé `x` (comme la lettre "x"), qui sert à répéter une chaîne un certain nombre de fois. Regardez :

Code : Perl

```
print "bla" x 10;
```

Code : Console

```
blablablablablablablablabla
```

Remarquez que cet opérateur n'est pas du tout le même que celui qui sert à multiplier deux nombres : c'est un détail qui a son importance !

Un autre opérateur utile est celui de concaténation, que l'on symbolise par un point (`.`). Concaténer deux chaînes de caractères signifie les coller bout à bout, comme ceci :

Code : Perl

```
my $chaine = 'cool' . 'ympique';  
print $chaine;
```

Code : Console

```
coolympique
```

Enfin, il est bien sûr possible de combiner ces deux opérateurs, comme cela :

Code : Perl

```
print 'tro' . 'lo' x 4;
```

Code : Console

```
trololololo
```

Do what I mean!

Maintenant que nous avons vu diverses manières de manipuler des scalaires, nous allons pouvoir nous pencher sur ce qui fait tout leur intérêt, à savoir le principe « *do what I mean* », qui est à la fois une spécificité et l'une des plus grandes forces de Perl.

Cette devise signifie « fais ce que je veux ! », par opposition à « fais ce que je dis ! ». Avec les scalaires, elle se traduit par le fait que Perl comprendra toujours (non, vraiment, TOUJOURS !) ce que vous avez *l'intention de faire*, sans que vous ayez besoin de le lui dire explicitement.

Démonstration, ouvrez grand vos mirettes :

Code : Perl

```
use strict;
use warnings;

my $quatre = "4";
my $deux = "2";
my $quatre_et_deux = $quatre . $deux; # concaténation
my $quatre_plus_deux = $quatre + $deux; # addition
my $quatre_x_deux = $quatre x $deux; # répétition
my $quatre_fois_deux = $quatre * $deux; # multiplication

print '$quatre_et_deux = ' . "$quatre_et_deux\n";
print '$quatre_plus_deux = ' . "$quatre_plus_deux\n";
print '$quatre_x_deux = ' . "$quatre_x_deux\n";
print '$quatre_fois_deux = ' . "$quatre_fois_deux\n";
```

Code : Console

```
$quatre_et_deux    = 42
$quatre_plus_deux  = 6
$quatre_x_deux     = 44
$quatre_fois_deux  = 8
```

Remarquez que les variables `$quatre` et `$deux`, telles que nous les avons définies, sont censées être des chaînes de caractères. Pour autant, nous avons pu aussi bien les additionner et les multiplier entre elles comme deux nombres que les

concaténer et les utiliser avec l'opérateur de répétition comme les chaînes de caractères qu'elles sont ; Perl n'a absolument pas bronché !

Ceci est dû au fait que les opérateurs de Perl sont très spécifiques. En utilisant l'opérateur `+`, on *sait très bien* ce que l'on demande à Perl : on lui demande d'additionner deux nombres. Et pourtant, les deux valeurs que nous lui avons données étaient des chaînes de caractères représentant des nombres. Qu'à cela ne tienne, au lieu de faire ce que nous lui avons *dit*, c'est-à-dire essayer d'appliquer une addition à deux chaînes (ce qui aurait inmanquablement planté dans pratiquement n'importe quel langage de programmation), Perl a fait ce que nous avions *l'intention de faire* : il a pris les devants, converti nos deux chaînes en nombres, et en a calculé la somme.

Il est très important de noter que ceci est rendu possible par le fait que Perl donne un sens plus fort *aux opérateurs* qu'aux types des variables qu'il manipule. **C'est l'opération que l'on utilise, ou plutôt, le contexte dans lequel on utilise les données, qui détermine leur type, et non l'inverse.** Très peu de langages traitent leurs données de cette façon, et Perl est le seul à le faire **parfaitement bien** (c'est-à-dire sans jamais provoquer de mauvaise surprise pour l'utilisateur, comme c'est parfois le cas de PHP).

Les autres langages cousins de Perl, comme par exemple Python, suivent en général la logique inverse : dans ces langages, c'est le type des données qui est le plus important, et qui décide de la signification des opérateurs.

Comprendre cette différence est *fondamental* pour bien programmer en Perl. C'est souvent à cause d'elle que naissent des disputes sans fin entre les partisans des langages fortement typés comme Python, et des langages faiblement typés comme Perl. La vérité à ce sujet, c'est que ni les uns, ni les autres n'ont raison. Ce qui importe n'est pas la force ni les caractéristiques du typage ; c'est avant tout que celui-ci s'inscrive dans une philosophie claire et cohérente. À ce titre, Python et Perl sont deux langages de grande qualité : chacun est fermement ancré dans une philosophie qui lui est propre et à laquelle il adhère jusque dans ses moindres détails. Le reste n'est véritablement qu'une question de confort et d'habitude. 😊

Place à la pratique !

On vous l'a dit, Perl est un langage conçu pour être **pratiqué**. Son intérêt réside dans le fait que plus vous l'utiliserez, plus vous trouverez en lui un outil commode et puissant. C'est pour cette raison que ce cours va contenir autant de théorie que d'exercices et de TP.

En ce qui nous concerne sur ce chapitre, il vous manque encore une toute petite notion à acquérir avant que vous puissiez écrire des programmes qui *font* quelque chose, et avec lesquels vous pourrez frimer devant les filles : la possibilité de leur donner un peu d'interactivité. Dans ce cours, ou du moins les chapitres qui viennent, celle-ci va passer par une petite instruction que nous avons rencontrée au tout début de ce chapitre, et qui ressemble à une espèce de diamant : `<>`.

Diamonds are a girl's best friend

Essayez le code suivant :

Code : Perl

```
use strict;
use warnings;

print "Comment vous appelez-vous ? ";
my $nom = <>; # Récupération du nom de l'utilisateur
chomp $nom;   # Retrait du saut de ligne
print "Bonjour, $nom !\n";
```

Ce programme attend que vous entriez votre nom avant de vous dire bonjour. Voici un exemple d'exécution :

Code : Console

```
Comment vous appelez-vous ? Obi-Wan Kenobi
Bonjour, Obi-Wan Kenobi !
```

La véritable signification de l'opérateur `<>`

Vous l'avez sûrement deviné, ce petit diamant ne fait pas que mettre le script en pause le temps d'appuyer sur Entrée, il sert aussi à récupérer ce que l'utilisateur a entré au clavier.

En fait, la vérité est encore plus vaste que cela. Le symbole `<>` est un élément de syntaxe servant à **lire une ligne dans un flux de données**. Tel quel, il s'agit d'un raccourci pour lire une ligne sur l'*entrée standard* du programme, ce qui devrait normalement s'écrire `<STDIN>` (essayez, et vous verrez que le programme ne changera pas de comportement).

Nous aurons l'occasion de revenir sur ce sujet (les flux de données et les entrées/sorties standards) à plusieurs reprises dans ce cours. Pour l'heure, retenez donc que *ce petit diamant sert à lire une ligne que l'utilisateur peut saisir au clavier*.

Chomp, chomp, chomp...

Derrière ce nom amusant se trouve une petite fonction très pratique, qui sert à retirer les sauts de ligne à la fin d'une chaîne de caractères. En effet, lorsque l'utilisateur saisit une donnée au clavier, il valide celle-ci en appuyant sur la touche Entrée. Cela explique la présence d'un passage à la ligne à la fin de la chaîne de caractères que l'on récupère. Cette fonction sert simplement à épurer la saisie de l'utilisateur, de façon à ne récupérer que le texte.

Pour rendre ces explications plus claires, regardez ce qui se passerait si l'on ne *chomp*-ait pas l'entrée de l'utilisateur :

Code : Perl

```
use strict;
use warnings;

print "Comment vous appelez-vous ? ";
my $nom = <>;
# chomp $nom;
print "Bonjour, $nom !";
```

Code : Console

```
Comment vous appelez-vous ? Raoul
Bonjour, Raoul
!
```

Vous voyez ?

Ainsi donc, repensez simplement à cette citation de Jean de LOLafontaine chaque fois que vous voudrez saisir une entrée de l'utilisateur : « Ah, vous m'écoutiez ? Eh bien *chomp*-ez maintenant ! ».

À vos claviers !

Il est maintenant temps pour vous de sortir vos mains de vos poches et de **coder**. Voici une petite série d'exercices qui va vous permettre d'appliquer tout ce que nous venons d'apprendre.

iPunish

Commençons par créer une petite application tendance et extrêmement pratique, destinée à écrire vos punitions à votre place.

Imaginez que vous ayez à copier 500 fois la phrase « *Je ne dois pas bavarder en classe.* ». Quelle galère ! Tout ce temps perdu alors que vous pourriez l'employer à ne rien faire... Grâce à **iPunish**, cette époque est maintenant révolue !

Entrez simplement une phrase et le nombre de fois que vous voulez la recopier, et iPunish fera le travail à votre place, tout cela dans le raffinement et l'élégance d'une interface en ligne de commande.

Regardez :

Code : Console

```
arnaud@netbook(~)% perl iPunish.pl
Entrez une phrase : Je ne recopierai plus mes punitions a la main.
Combien de copies ? 10
-----
Je ne recopierai plus mes punitions a la main.
Je ne recopierai plus mes punitions a la main.
Je ne recopierai plus mes punitions a la main.
Je ne recopierai plus mes punitions a la main.
Je ne recopierai plus mes punitions a la main.
Je ne recopierai plus mes punitions a la main.
Je ne recopierai plus mes punitions a la main.
Je ne recopierai plus mes punitions a la main.
Je ne recopierai plus mes punitions a la main.
Je ne recopierai plus mes punitions a la main.
```

Tout ce dont vous avez besoin pour réaliser cet exercice a été abordé dans ce chapitre, vous devriez donc être capables d'y arriver seuls.

Correction :

Secret ([cliquez pour afficher](#))

Code : Perl - iPunish.pl

```
use strict;
use warnings;

# Récupération de la phrase
# On ne la chompe pas, pour conserver le retour à la ligne
print "Entrez une phrase : ";
my $phrase = <>;

# Récupération du nombre de copies
print "Combien de copies ? ";
my $nb = <>;
chomp $nb;

# On affiche une jolie ligne pour séparer le résultat
# de la saisie
print '-' x 30 . "\n";

# Puis on recopie la phrase $nb fois
print $phrase x $nb;

# <>; # Pause de l'exécution sous Windows
```

Table de multiplication

Écrivez un script qui demande un nombre à l'utilisateur et récite sa table de multiplication comme ceci :

Code : Console

```
Entrez un nombre: 7
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
```

```
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
```

Vous allez vous apercevoir que ce code est très répétitif à écrire. Dans un prochain chapitre, nous découvrirons un (et même plusieurs) moyen(s) de le rendre beaucoup plus concis.

Correction :

Secret (cliquez pour afficher)

Code : Perl

```
use strict;
use warnings;

print "Entrez un nombre: ";
my $nb = <>;
chomp $nb;

print "$nb x 1 = " . $nb * 1 . "\n";
print "$nb x 2 = " . $nb * 2 . "\n";
print "$nb x 3 = " . $nb * 3 . "\n";
print "$nb x 4 = " . $nb * 4 . "\n";
print "$nb x 5 = " . $nb * 5 . "\n";
print "$nb x 6 = " . $nb * 6 . "\n";
print "$nb x 7 = " . $nb * 7 . "\n";
print "$nb x 8 = " . $nb * 8 . "\n";
print "$nb x 9 = " . $nb * 9 . "\n";
print "$nb x 10 = " . $nb * 10 . "\n";
```

Nous en avons terminé avec ce premier chapitre. Il est temps pour nous de résumer l'essentiel que vous devez en retenir :

- Un programme en Perl se présente comme une série **d'instructions**, servant à manipuler des **données**.
- Les instructions, en programmation, sont des **ordres** que l'on donne à la machine.
- Les données, dans l'ordinateur, ont des **types** différents.
- Perl est un **langage faiblement typé**. Cela signifie qu'il est plutôt tolérant si vous vous trompez de type.
- Ceci est dû au fait que Perl regroupe les types de données les plus élémentaires (nombres et chaînes de caractères) sous la notion de **scalaire**.
- En Perl, **c'est le contexte qui décide du type des données** qui sont manipulées et non l'inverse.

Il est possible que vous vous sentiez un petit peu assommé sous la quantité d'informations que vous venez d'emmagasiner. Prenez donc le temps de les digérer avant de passer à la suite.

Si vous avez bien suivi ce chapitre, les suivants devraient vous sembler beaucoup plus légers !

Les branchements conditionnels

Maintenant que nous avons appris les éléments de base du langage Perl, nous allons pouvoir, dès à présent, et ce jusqu'à la fin de cette première partie du cours, découvrir des mécanismes qui vont nous permettre d'écrire des scripts de plus en plus sophistiqués.

Dans ce chapitre, nous allons rendre nos scripts capables de prendre des **décisions**. Nous découvrirons à cette occasion :

- les scalaires permettant de manipuler des propositions logiques,
- différentes façons de représenter une condition,
- la notion de bloc de code,
- la notion de portée d'une variable.

Prédicats et booléens

Posons le contexte

Imaginons un programme dont l'exécution ressemblerait à ceci :

Code : Console

```
arnaud@netbook(perl)% perl password.pl
Entrez le mot de passe : sésame ouvre-toi
Accès refusé.
Allez vous faire voir !
```

```
arnaud@netbook(perl)% perl password.pl
Entrez le mot de passe : s'il te plait
Accès autorisé.
Bienvenue.
```

Il est évident que nous ne connaissons pas encore assez de Perl pour écrire un tel programme (à juste titre, puisque c'est l'objet de ce chapitre) mais avant de plonger le nez dans du code, essayons déjà de comprendre **ce qui se passe**.

Nous avons exécuté le même programme deux fois. Lors de la première exécution :

- Le programme demande à l'utilisateur de rentrer le mot de passe
- L'utilisateur entre le mot de passe : "sésame ouvre-toi"
- **Ce n'est pas le bon mot de passe**
- **Le programme envoie balader l'utilisateur**

Lors de la seconde exécution :

- Le programme demande à l'utilisateur de rentrer le mot de passe
- L'utilisateur entre le mot de passe : "s'il te plait"
- **C'est le bon mot de passe**
- **Le programme accueille l'utilisateur**

Nous voyons bien que le mot de passe entré par l'utilisateur **conditionne** le reste de l'exécution du programme : deux chemins sont possibles, ici symbolisés par la couleur du texte (rouge ou vert). Puisque nous avons sous les yeux tous les cas d'exécution possibles, nous pouvons en déduire *la structure logique* du programme :

- Le programme demande à l'utilisateur d'entrer son mot de passe
- L'utilisateur entre un mot de passe
- **Si le mot de passe est "s'il te plait"**
 - Le programme accueille l'utilisateur
- **SINON**
 - Le programme envoie balader l'utilisateur

Ceci met en évidence deux notions complètement nouvelles pour nous. La première, c'est bien évidemment cette structure « SI ... SINON ... ». On appelle cette structure un **branchement conditionnel**, et nous verrons un petit peu plus loin comment cela se traduit en Perl.

La seconde, que nous allons étudier tout de suite, c'est cette phrase colorisée en bleu : **le mot de passe est "s'il te plaît"**. En effet, cette phrase n'est pas un *ordre* puisqu'elle n'exprime pas une action ; il s'agit d'une proposition qui peut être soit *vraie*, soit *fausse*. Une telle proposition s'appelle un **prédicat**.

Les booléens

Comme on vient de le dire, un prédicat est une proposition qui peut être vraie ou fausse. La première chose que nous devrions chercher à savoir serait donc comment représenter le « vrai » ou le « faux » en programmation. En réalité, il n'y a rien de plus simple. Perl utilise pour cela les *booléens*.

Un booléen, en théorie, ce n'est rien d'autre qu'un objet qui peut prendre uniquement deux valeurs : *vrai* ou *faux*. Ce qu'il est important de remarquer en revanche, c'est qu'en Perl, les booléens ne sont pas un "type" à proprement parler. Il s'agit plutôt d'un *contexte* dans lequel on évalue les valeurs scalaires.

Ainsi, le nombre 0 en Perl est évalué comme étant *faux*, alors que toutes les autres valeurs numériques différentes de 0 sont évaluées comme *vraies*. De même, toutes les chaînes de caractères sont évaluées comme *vraies*, hormis la chaîne vide (""), ainsi que la chaîne "0".

Il existe enfin une valeur spéciale, *undef*, qui est évaluée comme étant *fausse*, mais nous en reparlerons plus tard.

Opérateurs de comparaison

Il est temps pour nous maintenant d'exprimer nos premiers prédicats.

Comparaison de nombres

C'est ici que le discours de ce début de chapitre va certainement avoir l'air compliqué pour pas grand chose. En effet, le terme « prédicat », que vous ne connaissiez peut-être pas, semble bien pompeux, quand on sait que vous en avez certainement déjà rencontré. Tenez, voici quelques exemples prédicats impliquant des comparaisons de nombres :

```
4 est supérieur à 3.           (vrai)
3 est supérieur à 4.           (faux)
10 est inférieur à 2.          (faux)
2 est inférieur ou égal à 10.   (vrai)
16 est égal à 16.              (vrai)
16 est différent de 2.         (vrai)
15 est supérieur ou égal à 16.  (faux)
15 est supérieur ou égal à 15.  (vrai)
```

Vous voyez ? Ce n'est vraiment pas sorcier ! Voici les opérateurs de Perl qui vont nous permettre d'exprimer ces prédicats :

Expression Perl	Prédicat
<code>\$a < \$b</code>	\$a est inférieur à \$b
<code>\$a > \$b</code>	\$a est supérieur à \$b
<code>\$a == \$b</code>	\$a est égal à \$b
<code>\$a != \$b</code>	\$a est différent de \$b
<code>\$a <= \$b</code>	\$a est inférieur ou égal à \$b
<code>\$a >= \$b</code>	\$a est supérieur ou égal à \$b

Voyons un peu ce que cela donne dans un script. Vous remarquerez que pour pouvoir afficher correctement les valeurs booléennes, nous sommes obligés de les convertir explicitement en nombres entiers, au moyen de la fonction `int`. Ainsi, ce code affichera 1 devant les propositions vraies, et 0 pour les fausses. C'est la façon la plus simple dont on puisse représenter un

booléen.

Code : Perl

```
use strict;
use warnings;

my $a = 2;
my $b = 4;

# La fonction "int" permet de convertir explicitement un scalaire
# en un nombre entier.

print "$a < $b : " . int ($a < $b) . "\n";
print "$a > $b : " . int ($a > $b) . "\n";
print "$a == $b : " . int ($a == $b) . "\n";
print "$a == $a : " . int ($a == $a) . "\n";
print "$a != $b : " . int ($a != $b) . "\n";
print "$a != $a : " . int ($a != $a) . "\n";
print "$a <= $b : " . int ($a <= $b) . "\n";
print "$a <= $a : " . int ($a <= $a) . "\n";
```

Code : Console

```
2 < 4 : 1
2 > 4 : 0
2 == 4 : 0
2 == 2 : 1
2 != 4 : 1
2 != 2 : 0
2 <= 4 : 1
2 <= 2 : 1
```

Comparaison de chaînes de caractères

Il n'y a pas que les nombres que nous pouvons comparer entre eux. Les chaînes de caractères, elles aussi, ont leurs opérateurs de comparaison...



Quoi ? Mais qu'est-ce que ça peut bien vouloir dire, « comparer » des chaînes de caractères ?

Posée comme cela, il est vrai que cette question ne semble pas évidente. Pourtant, il y a fort à parier que vous ayez déjà feuilleté un dictionnaire ou un annuaire dans votre vie, et que vous sachiez déjà qu'il existe un *ordre* dans lequel on peut classer les mots : **l'ordre alphabétique** (vous croiserez probablement aussi le terme d'ordre *lexicographique*, qui est plus général et ne s'applique pas seulement à des chaînes de caractères). Eh bien comparer deux chaînes de caractères, c'est simplement déterminer laquelle des deux vient en premier dans l'ordre alphabétique.

On dira donc qu'une chaîne *\$a* est *plus petite* qu'une chaîne *\$b* si *\$a* est rangée avant *\$b* dans l'ordre alphabétique.

Dès lors, nous pouvons en déduire tous les autres opérateurs, par analogie avec les comparaisons de nombres. En Perl ces opérateurs sont des mots-clés formés de façon mnémotechnique :

Expression	Prédicat	Signification
<i>\$a</i> lt <i>\$b</i>	<i>\$a</i> est plus petite que <i>\$b</i>	<i>lesser than</i> (plus petit)
<i>\$a</i> gt <i>\$b</i>	<i>\$a</i> est plus grande que <i>\$b</i>	<i>greater than</i> (plus grand)
<i>\$a</i> eq <i>\$b</i>	<i>\$a</i> est égale à <i>\$b</i>	<i>equals</i>

\$a ne \$b	\$a est différente de \$b	<i>not equal</i>
\$a le \$b	\$a est inférieure ou égale à \$b	<i>lesser or equal</i>
\$a ge \$b	\$a est supérieure ou égale à \$b	<i>greater or equal</i>

Voici quelques exemples d'application que vous êtes invités à compléter avec vos propres tests :

Code : Perl

```
use strict;
use warnings;

print "'cool' lt 'coolympique' : " . int('cool' lt 'coolympique') .
"\n";
print "'C' gt 'Perl' : " . int('C' gt 'Perl') . "\n";
print "'bla' eq 'bla' : " . int('bla' eq 'bla') . "\n";
print "'cool' ne 'coolympique' : " . int('cool' ne 'coolympique') .
"\n";
```

Code : Console

```
'cool' lt 'coolympique' : 1
'C' gt 'Perl' : 0
'bla' eq 'bla' : 1
'cool' ne 'coolympique' : 1
```

Il existe encore d'autres styles de prédicats que nous pouvons utiliser sur des chaînes de caractères que l'on appelle des **expressions régulières**, et qui sont emblématiques de Perl. Étant données leur puissance et la complexité de leur syntaxe, nous consacrerons, plus loin dans ce cours, un chapitre entier à leur étude.

Opérateurs logiques

La négation

La négation, désignée par l'opérateur unaire **!** en Perl, sert simplement à inverser une valeur booléenne :

Code : Perl

```
print "!0 : " . int (!0) . "\n";
print "!1 : " . int (!1) . "\n";
```

Code : Console

```
!0 : 1
!1 : 0
```

Le OU logique

Les logiciens sont des gens parfois difficiles à cerner. Voici ce qui pourrait être un dialogue entre deux d'entre eux :

— *Salut. J'ai une grande nouvelle. Ma femme vient d'accoucher, je suis papa !*
 — *Oh, félicitations ! C'est un garçon ou une fille ?*
 — *Absolument !*

... *bide* 🤔 ...

Cette boutade repose sur la signification du terme **OU** en logique. En français, la conjonction de coordination « ou » peut avoir plusieurs sens. Son utilisation la plus courante vise à proposer un choix entre plusieurs alternatives à notre interlocuteur. Ainsi, la question « Est-ce un garçon ou une fille ? » n'appelle que deux réponses possibles : soit « c'est un garçon », soit « c'est une fille » (quoique l'on pourrait envisager une troisième alternative, plutôt rare dans ce contexte : « ce n'est ni un garçon, ni une fille »...).

En revanche, dans la phrase « Je ne reste chez moi que lorsque le temps est à la pluie ou à la neige. », le *ou* ne représente pas un choix, mais prend ici le même sens que le **OU** logique. Ainsi cette phrase signifie que :

- S'il pleut mais qu'il ne neige pas, je préfère rester chez moi.
- S'il ne pleut pas mais qu'il neige, je préfère également rester chez moi.
- S'il ne pleut pas, et qu'il ne neige pas non plus, je ne reste pas chez moi (et tant pis s'il gèle).
- S'il pleut et qu'il neige en même temps, je reste chez moi.

Dans les phrases ci-dessus, on peut considérer que les propositions « il pleut » et « il neige » sont des prédicats. Le **OU** sert ici à combiner ces deux prédicats pour en former un seul et unique, que l'on va évaluer de la façon suivante :

Il pleut	Il neige	Il pleut OU il neige
FAUX	FAUX	FAUX
FAUX	VRAI	VRAI
VRAI	FAUX	VRAI
VRAI	VRAI	VRAI

Ainsi, on peut résumer le sens du **OU** logique, en disant que le prédicat « A **OU** B » est vérifié si **au moins l'une** des deux propositions A et B est vraie. Ceci explique notre boutade de tout à l'heure : en considérant que « c'est un garçon » et « c'est une fille » sont deux propositions logiques, on sait bien qu'au moins l'une des deux alternatives sera vraie, alors notre ami logicien a évalué que le prédicat « c'est un garçon **OU** une fille » était vérifié, d'où sa réponse. À titre purement informatif, on dit qu'une telle proposition logique qui ne peut jamais être fausse s'appelle une **tautologie**.

En Perl, nous n'avons pas **UN**, mais **DEUX** opérateurs qui représentent le **OU** logique (souvenez-vous : *TIMTOWTDI* !) : l'opérateur **|** et l'opérateur **or**. Regardez :

Code : Perl

```
print '0 or 0 : ' . (0 or 0) . "\n";
print '0 or 1 : ' . (0 or 1) . "\n";
print '1 or 0 : ' . (1 or 0) . "\n";
print '1 or 1 : ' . (1 or 1) . "\n";
print "\n";
print '0 || 0 : ' . (0 || 0) . "\n";
print '0 || 1 : ' . (0 || 1) . "\n";
print '1 || 0 : ' . (1 || 0) . "\n";
print '1 || 1 : ' . (1 || 1) . "\n";
```

Code : Console

```
0 or 0 : 0
0 or 1 : 1
1 or 0 : 1
```

```
1 or 1 : 1
0 || 0 : 0
0 || 1 : 1
1 || 0 : 1
1 || 1 : 1
```



Mais, mais... Mais pourquoi avoir créé deux syntaxes différentes pour le même opérateur ? C'est débile !

En réalité, ces deux opérateurs ne sont pas strictement identiques, comme nous le verrons un petit peu plus bas. Il est vrai que pour l'instant, à notre niveau de progression dans ce cours, cela ne semble pas très intéressant, mais nous aurons l'occasion, au fur et à mesure que nous avancerons, de trouver des usages divers et variés au OU logique, au travers desquels le choix de l'un ou l'autre de ces opérateurs aura de l'importance.



Pourquoi ne pas avoir utilisé la fonction `int` dans le précédent code ?

En voilà une question délicate !

C'est lié à la façon dont Perl évalue les expressions composées par le OU logique. Lorsque Perl rencontre une expression du type `$a or $b` (ou bien `$a || $b`), il va **d'abord** évaluer `$a` dans le contexte booléen. Si `$a` est évalué comme vraie, alors on sait déjà que `$a or $b` sera vraie, donc Perl donne directement à `$a or $b` la valeur de `$a` (sa vraie valeur scalaire, pas sa valeur booléenne) et il n'évalue pas `$b` (puisque c'est inutile). Par contre, si `$a` est fausse, alors Perl donne à cette expression la valeur de `$b` (sa vraie valeur scalaire, encore une fois).

Ce fonctionnement peut vous sembler quelque peu tarabiscoté. En réalité, vous n'avez pas besoin de le retenir pour le moment. Nous aurons l'occasion, là encore, d'y revenir en temps utile.

Le ET logique

Le ET logique sert à vérifier que deux propositions sont vraies à la fois. Ici, il n'y a pas d'ambiguïté sur le sens de la conjonction *et*. Il s'agit bien du même connecteur qu'en français. Ainsi, on peut analyser la proposition suivante « *Je suis attiré par les femmes qui sont jolies et intelligentes* » de façon logique, comme ceci :

- Si une femme est jolie et intelligente à la fois, alors elle est attirante du point de vue du locuteur.
- Si une femme est jolie mais pas intelligente, alors elle n'attire pas le locuteur.
- Si une femme n'est pas jolie, mais qu'elle est intelligente, alors elle n'attire pas le locuteur non plus.
- Si une femme n'est ni jolie, ni intelligente, alors cette fois encore, le locuteur ne la trouvera pas attirante.

On peut donc en déduire la *table de vérité* du ET logique :

Jolie	Intelligente	Jolie ET intelligente
FAUX	FAUX	FAUX
FAUX	VRAI	FAUX
VRAI	FAUX	FAUX
VRAI	VRAI	VRAI

Comme pour le OU logique, Perl dispose de deux opérateurs pour exprimer le ET : l'opérateur `&&` et l'opérateur `and`.

Code : Perl

```
print '0 and 0 : ' . (0 and 0) . "\n";
print '0 and 1 : ' . (0 and 1) . "\n";
print '1 and 0 : ' . (1 and 0) . "\n";
print '1 and 1 : ' . (1 and 1) . "\n";
print "\n";
print '0 && 0 : ' . (0 && 0) . "\n";
print '0 && 1 : ' . (0 && 1) . "\n";
print '1 && 0 : ' . (1 && 0) . "\n";
print '1 && 1 : ' . (1 && 1) . "\n";
```

Code : Console

```
0 and 0 : 0
0 and 1 : 0
1 and 0 : 0
1 and 1 : 1

0 && 0 : 0
0 && 1 : 0
1 && 0 : 0
1 && 1 : 1
```

En ce qui concerne l'évaluation de l'expression `$a and $b`, nous avons là aussi un fonctionnement similaire au OU logique : Perl évalue d'abord `$a` dans le contexte booléen. Si `$a` est fausse, alors inutile d'aller plus loin, l'expression `($a and $b)` prend la valeur scalaire de `$a`. Sinon, l'expression prend la valeur scalaire de `$b`.

Précédence des opérateurs logiques

Il est maintenant temps de répondre à la question que l'on a posée plus haut : qu'est-ce qui différencie les opérateurs `&&` et `||` des opérateurs `and` et `or` ?

La réponse tient en un mot : leur *précédence*. La précédence d'un opérateur, c'est sa *priorité*. Elle détermine l'ordre dans lequel les expressions sont évaluées.

Regardez :

Code : Perl

```
print '0 and 1 or 1 : ' . (0 and 1 or 1) . "\n";
print '0 and 1 || 1 : ' . (0 and 1 || 1) . "\n";
print '0 && 1 or 1 : ' . (0 && 1 or 1) . "\n";
print '0 and (1 or 1) : ' . (0 and (1 or 1)) . "\n";
print '(0 and 1) or 1 : ' . ((0 and 1) or 1) . "\n";
```

Code : Console

```
0 and 1 or 1 : 1
0 and 1 || 1 : 0
0 && 1 or 1 : 1
0 and (1 or 1) : 0
(0 and 1) or 1 : 1
```

Cet exemple illustre très bien le fait que les opérateurs `&&` et `||` ont une priorité beaucoup plus forte que les opérateurs `and` et `or`. Il vous est très fortement conseillé de dérouler mentalement ces expressions jusqu'à ce que vous en compreniez tous les tenants et les aboutissants.

Le « OU exclusif » (XOR)

Le dernier opérateur que nous verrons ici est le OU exclusif, que l'on note XOR. Pour dire les choses simplement, A XOR B est vraie si A est vraie OU BIEN B est vraie, mais pas les deux. En français, ce OU-là s'exprime souvent grâce à l'expression « ou bien », comme dans la phrase suivante : « Pour tenter un tel exploit, il faudrait être remarquablement brave, ou bien complètement stupide ».

Voici la table de vérité du XOR :

A	B	A XOR B
FAUX	FAUX	FAUX
FAUX	VRAI	VRAI
VRAI	FAUX	VRAI
VRAI	VRAI	FAUX

Contrairement aux deux précédents opérateurs logiques, le ou exclusif ne peut se traduire que d'une seule façon en Perl, au moyen de l'opérateur `xor`. Cette fois, nous sommes obligés d'utiliser la fonction `int` pour afficher le résultat des expressions, car il n'y a pas de mécanisme comparable aux deux précédents : toutes les expressions sont évaluées, il n'y aurait donc pas d'intérêt à ce que le résultat soit autre chose qu'un booléen.

Code : Perl

```
print '0 xor 0 : ' . int (0 xor 0) . "\n";
print '0 xor 1 : ' . int (0 xor 1) . "\n";
print '1 xor 0 : ' . int (1 xor 0) . "\n";
print '1 xor 1 : ' . int (1 xor 1) . "\n";
```

Code : Console

```
0 xor 0 : 0
0 xor 1 : 1
1 xor 0 : 1
1 xor 1 : 0
```

La fonction `defined`

La fonction `defined` permet de vérifier qu'une variable est définie. On dit qu'une variable est définie si une valeur lui a été affectée. Si ce n'est pas le cas, on dit que la variable n'est pas définie, et on symbolise sa valeur par la valeur spéciale `undef`. Un exemple valant mieux qu'un long discours, regardez plutôt :

Code : Perl

```
use strict;
use warnings;

my $var;      # $var est déclarée, mais elle n'est pas définie

print 'defined $var : ' . int (defined $var) . "\n";

$var = 42;    # $var est maintenant définie

print "\$var = $var \n";
```

```
print 'defined $var : ' . int (defined $var) . "\n";

$var = undef; # $var n'est maintenant plus définie

print "\$var = undef\n";
print 'defined $var : ' . int (defined $var) . "\n";
```

Code : Console

```
defined $var : 0
$var = 42
defined $var : 1
$var = undef
defined $var : 0
```

Exemple avancé

En combinant cette fonction avec le mécanisme d'évaluation des opérateurs logiques de Perl, on peut obtenir quelques comportements intéressants, bien que difficiles à lire au premier abord, comme celui-ci :

Code : Perl

```
use strict;
use warnings;

my $error = "\$var n'est pas définie.\n";

my $var;    # $var est déclarée, mais elle n'est pas définie
defined $var && print "\$var = $var\n" or print $error;

$var = 42;  # $var est maintenant définie
defined $var && print "\$var = $var\n" or print $error;

$var = undef; # $var n'est maintenant plus définie
defined $var && print "\$var = $var\n" or print $error;
```

Code : Console

```
$var n'est pas définie.
$var = 42
$var n'est pas définie.
```

La compréhension de ce code vous est laissée à titre d'exercice. Celle-ci n'est absolument pas indispensable pour suivre le reste de ce chapitre (on est déjà au-delà du niveau attendu), mais résume somme toute assez bien tout ce que nous avons vu jusqu'à maintenant. 😊

if, else, et unless**Où en sommes nous ?**

Reprenons, si vous le voulez bien, la structure logique du programme que nous avons imaginé au tout début de ce chapitre.

- Le programme demande à l'utilisateur d'entrer son mot de passe

- L'utilisateur entre un mot de passe
- **Si le mot de passe est "s'il te plait"**
 - Le programme accueille l'utilisateur
- **SINON**
 - Le programme envoie balader l'utilisateur

Nous savons maintenant exprimer le prédicat **le mot de passe est "s'il te plait"**, nous pouvons donc écrire une première version de ce programme.

Code : Perl

```
use strict;
use warnings;

print "Entrez le mot de passe : ";
my $pass = <>;
chomp $pass;

my $sacces = $pass eq "s'il te plait";
print '$sacces : ' . int($sacces) . "\n";
```

Code : Console

```
Entrez le mot de passe : sésame ouvre-toi
$sacces : 0
```

Code : Console

```
Entrez le mot de passe : s'il te plait
$sacces : 1
```

Il nous reste donc maintenant à exprimer en Perl cette structure « SI ... SINON ... ». Il suffit pour cela de parler un petit peu anglais.

if et else

En anglais, « si » se traduit par « *if* » et « sinon » se traduit par « *else* ». C'est pour cette raison que les principaux mots-clés servant à exprimer un branchement conditionnel sont **if** et **else**. Voici la structure générale d'un tel branchement :

Code : Perl

```
my $predicat; # ici le prédicat à tester

if ($predicat)
{
    # code à exécuter si le prédicat est vérifié
}
else
{
    # code à exécuter si le prédicat n'est pas vérifié
}
```

Comme vous le voyez, un branchement conditionnel s'exprime grâce à deux blocs de code, délimités par des accolades ({}). Plutôt que de nous attarder à décrire le fonctionnement de ces deux blocs, voici un exemple grâce auquel vous le comprendrez

directement :

Code : Perl

```
use strict;
use warnings;

print "Entrez le mot de passe : ";
my $pass = <>;
chomp $pass;

if ($pass eq "s'il te plait")
{
    # le mot de passe entré par l'utilisateur
    # est bien "s'il te plait"

    print "Accès autorisé.\n";
    print "Bienvenue.\n";
}
else
{
    # le mot de passe entré par l'utilisateur
    # n'est pas le bon

    print "Accès refusé.\n";
    print "Allez vous faire voir !\n";
}
```

Essayez-le, et vous constaterez qu'il s'agit bien du programme complet que nous avons imaginé au début.

Exercice

Écrivez un programme qui demande à l'utilisateur d'entrer la lettre H si c'est un homme, ou F si c'est une femme, et qui le salue en lui disant « Bonjour monsieur » ou « Bonjour madame » le cas échéant.

Branchements en série

Pour l'instant, nous avons le choix entre deux alternatives à gérer : soit le mot de passe est le bon, soit il ne l'est pas.

Imaginons maintenant que ce programme ait deux utilisateurs autorisés (donc deux mots de passe possibles). Le premier utilisateur, Jean, a pour mot de passe « s'il te plait », et le second utilisateur, Marie, a pour mot de passe « sesame ouvre-toi ». Comment coder ceci en Perl ?

Une première idée serait de faire quelque chose comme ceci :

Code : Perl

```
use strict;
use warnings;

print "Entrez le mot de passe : ";
my $pass = <>;
chomp $pass;

if ($pass eq "s'il te plait")
{
    # le mot de passe entré par l'utilisateur
    # est "s'il te plait"

    print "Accès autorisé.\n";
    print "Bienvenue, Jean.\n";
}
```



```

else
{
    if ($pass eq "sesame ouvre-toi")
    {
        print "Accès autorisé.\n";
        print "Bienvenue, Marie.\n";
    }
    else
    {
        print "Accès refusé.\n";
        print "Allez vous faire voir !\n";
    }
}

```

Comme vous le constatez, on teste d'abord si le mot de passe entré par l'utilisateur est celui de Jean, sinon, on teste si le mot de passe est celui de Marie, sinon, on envoie balader l'utilisateur. Cela fonctionne bien, certes, mais il existe une façon plus élégante d'écrire ce code en Perl. Ce "else if" peut être contracté au moyen du mot-clé **elsif**, comme cela :

Code : Perl

```

use strict;
use warnings;

print "Entrez le mot de passe : ";
my $pass = <>;
chomp $pass;

if ($pass eq "s'il te plait")
{
    print "Accès autorisé.\n";
    print "Bienvenue, Jean.\n";
}
elsif ($pass eq "sesame ouvre-toi")
{
    print "Accès autorisé.\n";
    print "Bienvenue, Marie.\n";
}
else
{
    print "Accès refusé.\n";
    print "Allez vous faire voir !\n";
}

```

C'est quand même un peu plus joli, non ? Imaginez ce que cela aurait pu donner si nous avions eu 15 utilisateurs différents !

Notez que l'on n'est pas obligé, ni dans ce cas, ni dans le précédent, de mettre une clause **else** à notre programme. On peut très bien écrire un bloc **if** tout seul : si le prédicat n'est pas satisfait, alors le programme ne fera rien :

Code : Perl

```

use strict;
use warnings;

print "Entrez le mot de passe : ";
my $pass = <>;
chomp $pass;

# si le mot de passe est le bon,
# on salue Jean
if ($pass eq "s'il te plait")
{
    print "Accès autorisé.\n";
    print "Bienvenue, Jean.\n";
}

```

```
}
# sinon, on ne fait rien
```

Exercice

Écrivez un programme qui demande à l'utilisateur d'entrer "F" si c'est une femme ou "H" si c'est un homme, et qui lui répond "Bonjour Monsieur" ou "Bonjour Madame" si c'est un homme ou une femme, ou bien "Salut machin" si la lettre entrée par l'utilisateur n'est ni "F" ni "H". Utilisez pour ce faire une clause **elsif**.

unless

Pour en finir avec les branchements conditionnels, examinons le dernier bloc que l'on peut utiliser : **unless**. En anglais, « *unless* » signifie « à moins que ». Il s'agit en fait de l'inverse de **if** en Perl.

Code : Perl

```
use strict;
use warnings;

print "Entrez le mot de passe : ";
my $pass = <>;
chomp $pass;

# Quitter le programme en envoyant balader l'utilisateur
# À MOINS QUE le mot de passe rentré soit le bon.

unless ($pass eq "s'il te plait")
{
    print "Accès refusé.\nAllez vous faire voir !\n";
    exit(1); # quitter le programme
}

print "Accès autorisé.\n";
print "Bienvenue.\n";

# ... suite du programme
```

Il est aussi possible d'adjoindre un bloc **else** à un bloc **unless** (ce qui permet d'exécuter du code si le prédicat évalué par **unless** n'est pas vérifié...), mais lorsque l'on y réfléchit bien, la structure "à moins que... sinon..." n'est pas très intuitive à comprendre. Pour cette raison, il est plutôt déconseillé d'utiliser conjointement **unless** et **else**, par soucis de lisibilité.

Travailler avec des blocs

Dans la précédente sous-partie, nous avons fait connaissance avec les blocs de code, délimités par des accolades. Profitons-en pour évoquer quelques bonnes pratiques à adopter lorsque l'on les utilise.

Mise en forme et indentation du code

Commençons par enfoncer des portes ouvertes. Un bloc, c'est d'abord une frontière qui permet de différencier le code qui est à l'intérieur de celui qui est à l'extérieur (nous allons voir en quoi dans un instant). À ce titre, il est indispensable de rendre ceci visible. Par convention, on met en évidence cette différence en **indentant** le code qui se trouve à l'intérieur du bloc. Indenter signifie *décaler le code vers la droite*. En général, ce décalage est d'une tabulation ou de 4 espaces, comme ceci :

Code : Perl

```
print "code à l'extérieur du bloc";

{
```

```
# cette ligne est indentée.  
print "code à l'intérieur du bloc";  
}
```

Pour la position des accolades, il existe plusieurs conventions, qui sont toutes acceptables. Dans ce cours, nous prendrons le parti de toujours aligner l'accolade ouvrante et même niveau que l'accolade fermante.

Ceci permet de rendre le code lisible, et il est primordial que vous preniez cette habitude dès maintenant. Constatez la différence par vous-même :

Code non indenté (**mauvais exemple**) :

Code : Perl

```
use strict;  
use warnings;  
  
print "Entrez le mot de passe : ";  
my $pass = <>;  
chomp $pass;  
  
if ($pass eq "s'il te plait")  
{  
    print "Accès autorisé.\n";  
    print "Bienvenue, Jean.\n";  
}  
else  
{  
    if ($pass eq "sesame ouvre-toi")  
    {  
        print "Accès autorisé.\n";  
        print "Bienvenue, Marie.\n";  
    }  
    else  
    {  
        print "Accès refusé.\n";  
        print "Allez vous faire voir !\n";  
    }  
}
```

Code bien indenté (**bon exemple**) :

Code : Perl

```
use strict;  
use warnings;  
  
print "Entrez le mot de passe : ";  
my $pass = <>;  
chomp $pass;  
  
if ($pass eq "s'il te plait")  
{  
    print "Accès autorisé.\n";  
    print "Bienvenue, Jean.\n";  
}  
else  
{  
    if ($pass eq "sesame ouvre-toi")  
    {  
        print "Accès autorisé.\n";  
        print "Bienvenue, Marie.\n";  
    }  
}
```

```
    }  
    else  
    {  
        print "Accès refusé.\n";  
        print "Allez vous faire voir !\n";  
    }  
}
```

Présentation alternative (**bon exemple**) :

Code : Perl

```
use strict;  
use warnings;  
  
print "Entrez le mot de passe : ";  
my $pass = <>;  
chomp $pass;  
  
if ($pass eq "s'il te plait") {  
    print "Accès autorisé.\n";  
    print "Bienvenue, Jean.\n";  
}  
else {  
    if ($pass eq "sesame ouvre-toi") {  
        print "Accès autorisé.\n";  
        print "Bienvenue, Marie.\n";  
    }  
    else {  
        print "Accès refusé.\n";  
        print "Allez vous faire voir !\n";  
    }  
}
```

Niveau lisibilité, il n'y a pas photo...

Notion de portée lexicale

Cette notion est extrêmement importante. Jusqu'à présent, lorsque nous travaillions sans placer de code dans des blocs, nous définissions toutes nos variables « sur le même plan ». Or lorsque l'on définit une variable dans un bloc, on doit se soucier de sa portée, c'est-à-dire des endroits dans le code où elle *existe*.

Pour faire simple, une variable existe dans le bloc dans lequel elle a été définie, et dans les blocs contenus à l'intérieur de celui-ci, mais pas à l'extérieur. On dira donc, pour bien faire, que *la portée lexicale d'une variable est locale au bloc dans lequel elle est déclarée*.

Voyons cela au travers d'un exemple.

Code : Perl

```
use strict;  
use warnings;  
  
my $var = "Hello";  
  
{  
    my $var2 = "World";  
    print $var; # affiche "Hello".  
    print $var2; # affiche "World".  
}
```

```
print $var; # affiche "Hello".
print $var2; # /\ ERREUR, Perl va vous envoyer ballader car $var2
n'existe plus
           # et le script ne sera pas exécuté.
```

Il est aussi important de noter que nous devons ce comportement (cette sécurité, en fait) au mode `strict` de Perl. En effet, si nous n'avions pas invoqué le pragma `strict` au début de ce programme, l'instruction `print $var2` aurait simplement été évaluée comme l'affichage d'une variable que l'on déclare dans la foulée, et n'aurait donc rien affiché (en toute rigueur, elle aurait affiché la représentation sous forme d'une chaîne de la valeur `undef`, c'est-à-dire rien du tout).

Ainsi le programme n'aurait pas planté, mais le *warning* suivant serait quand même apparu :

Code : Console

```
Use of uninitialized value $var2 in print at test.pl line 13.
```

Et bien sûr, si nous avions aussi oublié d'activer les warnings au moyen de l'instruction `"use warnings;"`, ce warning ne serait pas apparu non plus, le code se serait exécuté, et cette ligne n'aurait rien affiché sans aucune explication, d'où l'importance de ces deux pragmas !

La syntaxe « en ligne »

Et si on commençait à rendre nos codes un peu plus *perlusques* ?

Cette sous-partie est ici à titre d'exhaustivité, car nous trouverons, pour l'instant, relativement peu de cas dans lesquels utiliser cette syntaxe « en ligne ».

Il arrive que dans un code, on écrive un bloc `if` ou un bloc `unless` qui tienne en une seule ligne, comme ceci :

Code : Perl

```
use strict;
use warnings;

print "Entrez le mot de passe : "; # eh oui, encore cet exemple !
my $pass = <>;
chomp $pass;

# À moins que le mot de passe soit "s'il te plait"
unless ($pass eq "s'il te plait")
{
    # Quitter en affichant une erreur
    die "Erreur d'authentification";
}

# Déroulement normal du programme.
print "Bienvenue\n";

# etc.
```

Ici, la fonction `die`, qui est nouvelle pour vous (et qui signifie "meurs") sert à déclencher une erreur dans le programme. Si cette erreur n'est pas rattrapée (nous verrons comment faire dans un autre chapitre), alors le programme se termine en affichant l'erreur donnée en argument à `die`.

Comme vous le constatez, ce bloc prend visuellement pas mal de place, d'autant que sa fonction est plutôt simple : quitter si le mot de passe est erroné.

Lorsqu'il s'agit d'un tel bloc `if` ou `unless`, sans clause `else`, on peut l'écrire en une seule ligne, et ce sans perdre en lisibilité, comme cela :

Code : Perl

```

use strict;
use warnings;

print "Entrez le mot de passe : ";
my $pass = <>;
chomp $pass;

# Quitter À MOINS QUE le mot de passe soit "s'il te plait".
die "Erreur d'authentification" unless $pass eq "s'il te plait";

# alternativement, on aurait pu écrire ceci :

# Quitter SI le mot de passe n'est pas "s'il te plait".
die "Erreur d'authentification" if $pass ne "s'il te plait";

# Suite du programme
print "Bienvenue\n";

# etc.

```

Comme vous le voyez, on ajoute simplement une condition à l'exécution de la ligne, et les parenthèses entourant le prédicat ne sont plus nécessaires. Il s'agit véritablement de sucre syntaxique, mais vous remarquerez que le code se lit finalement exactement de la même façon que sa signification, ce qui permet de ne pas surcharger le programme visuellement, tout en gardant une syntaxe claire et lisible.

TP : jeu de devinettes (première version)

Afin d'appliquer ce que nous avons appris dans ce chapitre, nous allons, en guise de TP corrigé, créer un petit jeu de devinettes tout simple que nous améliorerons plus tard.

L'utilisateur doit choisir un animal parmi une liste donnée. Le programme va ensuite poser des questions à l'utilisateur, auxquelles celui-ci devra répondre par oui (o) ou non (n), jusqu'à ce qu'il ait deviné de quel animal il s'agit.

Voici deux exemples d'exécution :

Code : Console

```

Choisissez un animal parmi les suivants :
Chat, Mouette, Lama, Chouette
-----
Est-ce que c'est un oiseau [o/n] ? n
Est-ce que cet animal crache [o/n] ? o
C'est un lama !

```

Code : Console

```

Choisissez un animal parmi les suivants :
Chat, Mouette, Lama, Chouette
-----
Est-ce que c'est un oiseau [o/n] ? o
Est-ce que c'est un oiseau nocturne [o/n] ? o
C'est une chouette !

```

Bien entendu, pour réussir, vous allez devoir appliquer ce que nous avons vu dans ce chapitre, et utiliser des blocs **if/else**.

Dans un prochain TP, nous ferons en sorte que la liste puisse augmenter et que le jeu soit capable d'apprendre à deviner de nouvelles entrées au fur et à mesure que l'on y joue.

Bon courage ! 🤖

Correction

Secret (cliquez pour afficher)

Code : Perl

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

# Jeu de devinettes
# -----
# L'utilisateur choisit un animal parmi une liste
# donnée, et le programme doit deviner duquel il s'agit.

use strict;
use warnings;

print "Choisissez un animal parmi les suivants :\n";
print "Chat, Mouette, Lama, Chouette\n";
print '-' x 30 . "\n";

print 'Est-ce que c\'est un oiseau [o/n] ? ';
my $choice = <>;
chomp $choice;

# la fonction 'lc' sert à convertir une chaîne lettres en
# minuscules
# lc : lower case
if (lc($choice) eq 'o')
{
    # c'est un oiseau
    print 'Est-ce que c\'est un oiseau nocturne [o/n] ? ';
    $choice = <>;
    chomp $choice;
    if (lc($choice) eq 'o')
    {
        print "C'est une chouette !\n";
    }
    else
    {
        print "C'est une mouette !\n";
    }
}
else
{
    print 'Est-ce que cet animal crache [o/n] ? ';
    $choice = <>;
    chomp $choice;
    if (lc($choice) eq 'o')
    {
        print "C'est un lama !\n";
    }
    else
    {
        print "C'est un chat !\n";
    }
}

# <>;
```

Vous pouvez souffler ! Nous sommes arrivés au bout de ce chapitre.

Résumons donc les notions essentielles que nous venons de voir :

- Contrairement à une instruction, un **prédicat** n'est pas un ordre, mais une proposition qui peut être vraie ou fausse.
- On peut construire des prédicats en comparant des scalaires entre eux.
- La valeur de vérité d'un prédicat est une donnée **booléenne**.
- Les booléens peuvent être combinés entre eux au moyen d'**opérateurs logiques**.
- Grâce à la construction **if/else**, un programme peut prendre des décisions et exécuter du code différent selon les **conditions** dans lesquelles il se trouve.
- Les variables définies avec le mot-clé **my** en Perl ne sont pas accessibles partout dans le code : celles-ci ont **une portée locale** au bloc dans lequel elles sont définies.

Comme vous le voyez, nos programmes commencent à ressembler à quelque chose, puisque leur exécution peut maintenant passer plusieurs chemins différents en fonction des décisions qu'ils prennent. Dans le prochain chapitre, nous découvrirons un autre style de blocs qui nous permettra d'effectuer des tâches répétitives à l'infini.

Processus itératifs et boucles

Dans le chapitre précédent, vous avez découvert vos premières **structures de contrôle de flux** avec les constructions **if/else** et **unless**. À présent, nous allons poursuivre notre lancée en travaillant sur un autre type de structure de contrôle : les boucles.

Comme vous le savez sûrement, le principal avantage des programmes et des robots sur nous est qu'ils peuvent répéter inlassablement les mêmes actions sans jamais se tromper ni se fatiguer. C'est précisément ce que nous allons étudier : comment faire en sorte que vos programmes se répètent.

Dans ce chapitre vous apprendrez :

- À caractériser un processus itératif,
- À créer une boucle **while** en Perl,
- À ouvrir, lire, écrire et fermer des fichiers sur votre ordinateur,
- À utiliser d'autres types de boucles (**unless**, **do ... while** et **for**) et à les choisir selon vos besoins.

Allez, on s'y colle !

La boucle de base : **while**

De la même façon que dans le chapitre précédent, commençons par examiner l'exécution d'un programme afin d'illustrer le comportement nouveau que nous allons étudier dans ce chapitre.

Code : Console

```
Je vais compter les lamas pour m'endormir.  
Jusqu'à combien dois-je compter ? 13  
1 lama  
2 lamas  
3 lamas  
4 lamas  
5 lamas  
6 lamas  
7 lamas  
8 lamas  
9 lamas  
10 lamas  
11 lamas  
12 lamas  
13 lamas  
ZZZZzzzzzzzz...
```

En effet, aussi bête que cela en a l'air, nous ne sommes pas encore capables, avec nos connaissances actuelles, d'écrire un vrai programme capable de compter jusqu'à une valeur arbitraire... Rassurez-vous cependant, car nous allons combler cette lacune très vite !

Pour cela, commençons par comprendre ce qu'il se passe lorsque l'on compte jusqu'à 13.

D'abord, supposons que l'on travaille avec une variable que l'on va nommer `$compteur`. Au tout début, on initialise ce compteur avec la valeur 1, puis on énonce « 1 lama ». Ensuite, c'est là que les choses se corsent. **Tant que** notre compteur n'est pas arrivé à 13, on le fait « avancer d'un cran » (dans le jargon des programmeurs, on dit que l'on **incrémente** sa valeur), puis on énonce « `$compteur` lamas ». Dans notre exemple, cette opération est **répétée** 12 fois. Enfin, une fois que l'on est arrivés à 13, on s'arrête.

Cela se résume en pseudo-code comme ceci :

- `$compteur = 1`
- `print "$compteur lama"`
- **Tant que** `$compteur < 13`:
 - `$compteur = $compteur + 1`
 - `print "$compteur lamas"`
- `print "ZZZZzzzzzz..."`

Comme vous le voyez, nous sommes déjà pratiquement capables d'écrire ce programme entièrement en Perl, à l'exception d'un détail : cette construction **Tant que**. Celle-ci sert à décrire ce que l'on appelle un **processus itératif**, c'est-à-dire un processus dans lequel une même opération est réalisée de façon *répétitive et progressive*. Plus précisément, ce « tant que » permet de décrire **une boucle**.

Considérons la phrase suivante :

« Il insistera tant qu'elle ne l'aura pas giflé. »

On peut découper cette phrase en trois éléments :

Il insistera **tant qu'elle ne l'aura pas giflé**.

En **vert**, nous avons l'expression d'une *action*. En **rouge**, nous avons la clé de la construction de cette phrase, à savoir la locution « tant que ». En **bleu**, nous avons une condition, ou plutôt un *prédicat*. Intuitivement, nous savons, à la lecture de cette phrase, que l'**action** va être répétée encore et encore, « en boucle », jusqu'au moment où... « PAF ! », le **prédicat** ne sera plus vérifié, auquel cas la boucle sera rompue, et l'action s'arrêtera pour passer à autre chose.

En Perl, une telle boucle peut être créée en traduisant simplement ce « Tant que » par son synonyme anglais, « while » :

Code : Perl

```
while ($predicat) # Tant que le prédicat est vérifié
{
    action ();    # Répéter la même action en boucle
}
```

Nous sommes donc en mesure d'écrire le code de l'exemple de tout à l'heure :

Code : Perl

```
use strict;
use warnings;

print "Je vais compter les lamas pour m'endormir.\n";

my $limite;
print "Jusqu'à combien dois-je compter ? ";
$limite = <>;

# Situation initiale, le compteur est à 1
my $compteur = 1;
print "$compteur lama\n";

# Tant que le compteur n'a pas atteint sa limite
while ($compteur < $limite)
{
    # On compte
    $compteur = $compteur + 1;
    print "$compteur lamas\n";
}

print "ZZZZZZzzzzzz...\n";
```

Ordre d'exécution

Comme vous le constatez, nous commençons à faire face à des programmes qui ne suivent plus forcément l'ordre du fichier source, mais sont capables de « revenir en arrière » de quelques lignes pour se répéter. À ce titre, il est bon de fixer dès maintenant dans votre esprit la façon dont une boucle **while** est exécutée par perl.

Celle-ci ne diffère pas de la façon dont les boucles sont exécutées dans les autres langages impératifs :

- Lorsque l'interpréteur arrive au niveau du **while**, celui-ci commence par évaluer le prédicat.
- Si ce dernier est vérifié, l'interpréteur exécute *tout le contenu du bloc* (à moins de tomber sur une instruction spéciale, mais nous y reviendrons au moment voulu), sinon, il passe son chemin et reprend l'exécution après le bloc.
- Une fois le bloc exécuté, l'interpréteur va évaluer à nouveau le prédicat.
- Si le prédicat est encore vérifié, l'interpréteur va exécuter à nouveau tout le contenu du bloc.
- ...

Cela peut sembler idiot à rappeler, mais il s'agit d'un point sur lequel beaucoup de débutants commettent des erreurs ; on ne sort de la boucle que lorsque le bloc a fini d'être exécuté, et que le prédicat n'est plus vérifié.

Code : Perl

```
my $var = 12;
while ($var != 13)
{
    $var = 13;          # à ce niveau le prédicat n'est plus vérifié

    print "$var\n";     # mais cette ligne sera quand même exécutée
} # c'est seulement une fois rendu ici, à la fin du bloc,
  # que le prédicat sera à nouveau évalué
  # et que l'on sortira de la boucle.
```

Opérateurs d'incrément et de décrémentation

Dans l'exemple du début de ce chapitre, on trouve l'instruction suivante.

Code : Perl

```
$compteur = $compteur + 1;
```

En programmation impérative, on utilise très souvent ce genre d'instructions, que l'on appelle une incrément. C'est la raison pour laquelle Perl dispose d'une syntaxe plus légère pour effectuer cette opération :

Code : Perl

```
$compteur = 4;
$compteur += 1; # 5
$compteur += 2; # 7
$compteur -= 1; # 6
$compteur /= 2; # 3
$compteur .= " petits cochons"; # "3 petits cochons"
```

D'ailleurs, dans les cas très particuliers où l'on souhaite simplement ajouter ou retrancher la valeur 1 à une variable, il existe même une syntaxe encore plus courte, consistant à utiliser les opérateurs unaires d'incrément (**++**) ou de décrémentation (**--**) :

Code : Perl

```
$compteur = 4;  
$compteur++; # 5  
++$compteur; # 6  
$compteur--; # 5  
--$compteur; # 4
```



Quelle est la différence entre `$compteur++` et `++$compteur` ?

Il faut savoir qu'en Perl, toutes les instructions retournent une valeur. La différence entre ces deux syntaxes est que lorsque l'opérateur est situé *après* la variable, l'instruction prend la valeur de la variable avant que celle-ci soit incrémentée. À l'inverse, lorsque l'opérateur est situé *avant* la variable, l'instruction prend la valeur de la variable après incrémentation :

Code : Perl

```
my $compteur = 4;  
print $compteur++; # affiche 4  
print $compteur;   # affiche 5  
print ++$compteur; # affiche 6
```

TP : jeu du plus ou moins

Avant de poursuivre, passons dès maintenant à la pratique et examinons un programme dont le fonctionnement repose sur une boucle while.

Nous allons programmer le célèbre jeu du *plus ou moins*. Il s'agit d'un petit jeu très simple, dont le but est de deviner un nombre choisi au hasard entre 1 et 100 par l'ordinateur :

Code : Console

```
Devinez le nombre mystère.  
> 50  
C'est moins.  
> 25  
C'est moins.  
> 12  
C'est plus.  
> 18  
C'est plus.  
> 21  
Gagné !
```

Pour programmer ce petit jeu, nous devons commencer par l'analyser, étape par étape. Voici ce que cela devrait donner :

- L'ordinateur choisit un nombre aléatoirement entre 0 et 100.
- Tant que le joueur n'a pas trouvé le nombre mystère :
 - Le joueur entre un nombre
 - Si le nombre est plus grand que le nombre mystère, l'ordinateur répond « C'est moins ».
 - Sinon, s'il est plus petit que le nombre mystère, l'ordinateur répond « C'est plus ».
- Une fois que le joueur a trouvé le nombre mystère, la partie est terminée.

Maintenant que l'algorithme du programme est défini, il ne nous reste plus qu'à trouver comment coder chaque étape.

Tirer un nombre au hasard

Pour générer une valeur aléatoire, en Perl, nous allons utiliser la fonction `rand`. Celle-ci, lorsqu'on l'appelle sans lui donner d'argument, retourne aléatoirement un nombre décimal compris entre 0 et 1.

On peut aussi lui passer un argument, pour choisir un nombre au hasard entre 0 et une valeur donnée (exclue) :

Code : Perl

```
my $val = rand 10 # retourne un nombre compris entre 0 et 9.999999
...
```

Dans notre cas, puisque nous voulons choisir au hasard un nombre entier compris entre 0 et 100, il nous suffira donc d'appeler `rand` en fixant sa limite à 101, et de convertir le résultat en un nombre entier, avec la fonction `int`, que nous avons rencontrée dans le chapitre précédent, comme ceci :

Code : Perl

```
my $nb_mystere = int (rand 101); # Tirage aléatoire d'un nombre
                                # compris entre 0 et 100
```

La boucle principale

Une fois que nous avons choisi un nombre mystère, il faut que l'utilisateur essaye de le deviner. Pour cela, nous allons créer une boucle : tant que le joueur n'aura pas trouvé le nombre mystère, on lui donnera des indices pour l'y aider.

Code : Perl

```
# /\ N'EXÉCUTEZ PAS CE SCRIPT TEL QUEL /\

use strict;
use warnings;

my $nb_mystere = int (rand 101);
my $essai = -1;

print "Devinez le nombre mystère.\n";

while ($essai != $nb_mystere)
{
    # Le joueur essaye de deviner le nombre mystère
}

# Si l'on arrive ici, c'est que le joueur a gagné.
print "Gagné !"
```

Il y a plusieurs remarques à faire sur ce code. *Tout d'abord, ne l'exécutez pas tel quel !* En effet, étant donné que nous n'avons encore rien placé dans le corps de la boucle, la valeur de `$essai` ne sera jamais modifiée, et le programme va tourner dans le vide indéfiniment. Il vous faudrait alors, pour pouvoir terminer ce programme, l'interrompre en appuyant simultanément sur les touches Ctrl + C, de façon que le processus soit tué par votre système d'exploitation.

On remarque aussi que la variable `$essai`, qui va servir à récupérer les tentatives du joueur, est initialisée à -1. De cette façon, lorsque le prédicat du `while` sera évalué pour la première fois, celui-ci sera vérifié, ce qui permettra au programme d'entrer dans la boucle.

Programme final

Il ne nous reste plus qu'à coder le déroulement d'un tour du jeu, ce qui n'est pas compliqué en soi : on récupère dans la variable `$essai` l'entrée de l'utilisateur, puis on compare cette valeur au nombre mystère. Il s'agit d'une simple clause `if/elsif`, comme nous en avons croisé dans le précédent chapitre :

Code : Perl

```
use strict;
use warnings;

my $nb_mystere = int (rand 101);
my $essai = -1;

print "Devinez le nombre mystère.\n";

while ($essai != $nb_mystere)
{
    print '> ';
    $essai = <>;

    if ($essai < $nb_mystere)
    {
        print "C'est plus.\n";
    }
    elsif ($essai > $nb_mystere)
    {
        print "C'est moins.\n";
    }
}

print "Gagné !";
```

Vous remarquerez que l'on ne `chomp`-e pas, dans ce code, l'entrée de l'utilisateur. En effet, puisque la variable `$essai` va être comparée au nombre mystère au moyen des opérateurs `<` et `>`, celle-ci sera automatiquement convertie en nombre, opération pendant laquelle le `"\n"` en fin de ligne sera ignoré.

Traitement de fichiers

L'un des usages les plus courants de la boucle `while` en Perl est lorsque le programme doit lire ou écrire dans un fichier sur le disque dur. Nous allons donc, dans les prochains exemples, apprendre à interagir de façon basique avec des fichiers texte. Comme vous allez le voir, ce n'est véritablement pas sorcier, mais cela vous permettra de créer dès maintenant des scripts utiles et intéressants !

open et close

Que ce soit pour lire ou pour écrire dans un fichier, il est évident qu'il faut que nous soyons capables de les ouvrir. Pour ce faire, Perl nous propose la fonction `open`. Un petit coup d'œil sur [la documentation de cette fonction](#) vous fera comprendre très rapidement qu'il existe moult façons de l'utiliser. Afin de ne pas surcharger ce chapitre inutilement, nous nous contenterons d'examiner ici son usage le plus commun, c'est-à-dire la syntaxe utilisant trois arguments :

```
open FILEHANDLE, MODE, EXPR
```

où :

- `FILEHANDLE` est... un *filehandle* (littéralement, une « *poignée sur un fichier* »), c'est-à-dire ni plus ni moins qu'une **variable** un peu spéciale qui représente le fichier avec lequel on interagit. Nous y reviendrons.
- `MODE` est une chaîne de caractères qui décrit l'usage que l'on va faire de ce fichier (lecture, écriture, ajout...).
- `EXPR` est une autre chaîne de caractères qui contient **le chemin du fichier**.

En ce qui concerne le `MODE`, nous allons nous contenter des trois plus courants pour ce chapitre. Vous aurez probablement l'occasion d'en rencontrer d'autres, beaucoup plus subtils et moins usités, dans la suite de ce cours, ou simplement en consultant la documentation en fonction de vos besoins. Ces trois `MODEs` sont en fait rigoureusement identiques à ce que vous pourrez

rencontrer dans la syntaxe d'un shell Unix :

Mode	Signification
'<'	Ouvrir le fichier en lecture. L'appel à <code>open</code> échoue si le fichier n'existe pas.
'>'	Ouvrir le fichier en écriture. L'appel à <code>open</code> échoue si l'utilisateur n'a pas les droits d'accès sur ce fichier ou son dossier. Si le fichier n'existe pas, un nouveau est créé. Si le fichier existe, son contenu précédent est effacé.
'>>'	Ouvrir le fichier en mode "ajout". Sensiblement la même chose que le mode écriture. Si le fichier existe déjà, tout ce que l'on écrira dedans sera placé à la suite du contenu actuel.



N'est-il pas possible d'ouvrir un fichier à la fois en lecture et en écriture ?

Dans l'absolu, si. Néanmoins, il est très peu probable que vous en ayez réellement besoin dans un programme, et il s'agit d'une pratique qui peut s'avérer dangereuse pour vos fichiers si vous ne savez pas exactement ce que vous êtes en train de faire. C'est la raison pour laquelle nous ferons l'impasse sur ce sujet dans ce cours. Les plus téméraires d'entre vous pourront toujours consulter la documentation de Perl pour trouver comment faire.

Les chemins des fichiers, quant à eux, dépendent de votre système d'exploitation. Ainsi, sous Windows, les dossiers doivent être séparés par des antislashes, il faut donc faire attention à les échapper, ainsi que les espaces ("`C:\\Mes\\ Documents\\Perl\\fichier_test.txt`"). Sous les OS basés sur Unix, en revanche, les dossiers sont classiquement séparés par des slashes ("`/home/arnaud/perl/fichier_test.txt`"), ce qui rend la tâche plus aisée.



D'une façon générale, il est conseillé d'éviter de créer des chemins ou des fichiers dont les noms contiennent des espaces ou des caractères spéciaux/accents.

Enfin, pour fermer un fichier, une fois que vous avez fini de travailler avec, vous pouvez utiliser la fonction `close` en lui passant en argument le *filehandle* du fichier que vous avez ouvert.

Résumons-nous dans un exemple :

Code : Perl

```
# Chemin du fichier à ouvrir en écriture.
# my $filename = "C:\\fichier_test.txt"; # Sous Windows
my $filename = "/home/arnaud/fichier_test.txt"; # Sous Linux

my $fh;
open ($fh, '>', $filename) or die "Impossible d'ouvrir le fichier
$filename en écriture";
# ... code qui écrit dans le fichier ...
close $fh;

open ($fh, '<', $filename) or die "Impossible d'ouvrir le fichier
$filename en lecture";
# ... code qui lit le fichier ...
close $fh;
```

Remarquez que le résultat de la fonction `open` est évalué comme faux lorsque l'ouverture du fichier échoue. Ceci nous permet de rajouter la petite clause `or die "..."`, qui sert à quitter le programme en affichant un message d'erreur en cas d'échec. Alternativement, vous pouvez vérifier que l'ouverture a bien fonctionné en testant le filehandle avec la fonction `defined`. Si l'ouverture a échoué, alors la variable `$fh` (dans l'exemple) n'est pas définie.

Lecture et écriture

Jusqu'ici, pour afficher du texte dans la console, nous avons utilisé la fonction `print`. Eh bien dorénavant, pour écrire des données dans un fichier, nous utiliserons... aussi la fonction `print` ! Regardez plutôt :

Code : Perl

```
use strict;
use warnings;

open (my $fh, '>', 'fichier_test.txt') or die "Impossible d'ouvrir
le fichier";
print $fh "Hello, World !\n"; # On précise à la fonction print le
                             # filehandle dans lequel on désire
écrire
close $fh;
```

Si vous lancez ce script, celui-ci va créer un fichier appelé `fichier_test.txt`, et écrire la chaîne `"Hello, World !"` à l'intérieur. Vérification sous Linux :

Code : Console

```
arnaud@laptop(perl)% perl helloworld_fichier.pl
arnaud@laptop(perl)% cat fichier_test.txt
Hello, World !
```

Vous remarquerez que pour afficher le contenu du fichier dans cet exemple, on a utilisé le programme `cat`, que tous les utilisateurs d'Unix connaissent bien, et qui sert simplement à afficher le contenu d'un fichier, ou bien à envoyer sur la sortie standard le texte qu'il reçoit en entrée. Dans un instant, nous reprogrammerons cet utilitaire en une seule ligne de Perl. 😊

Revenons à notre code : vous n'aurez pas manqué de remarquer que nous avons simplement précisé à la fonction `print` le *filehandle* dans lequel nous désirions écrire. En effet, jusqu'à maintenant, nous n'avions utilisé cette fonction que dans le cas particulier où l'on ne précise aucun *filehandle* à `print`, auquel cas celle-ci réagit simplement en écrivant **sur le flux de sortie standard** du programme, c'est-à-dire, par défaut, dans la console. Écrire dans un fichier est donc in fine tout aussi simple que d'écrire dans la console : il suffit de dire à Perl où envoyer les données.

Pour lire les données d'un fichier, vous n'allez pas franchement vous sentir dépaysés non plus. Vous connaissez déjà l'opérateur qui va nous permettre de le faire, puisque vous l'avez déjà utilisé, mais, cette fois encore, vous ne l'avez utilisé jusqu'à maintenant que dans le cas où vous ne précisez pas à Perl où vous vouliez qu'il lise des données :

Code : Perl

```
use strict;
use warnings;

open (my $fh, '<', 'fichier_test.txt') or die "Impossible d'ouvrir
le fichier";
my $ligne = <$fh>; # Eh ouais, les diamants sont éternels !
close $fh;

print $ligne;
```

En effet, le petit diamant que nous utilisons pour lire une ligne entrée par l'utilisateur dans la console (plus précisément, pour lire une ligne **sur l'entrée standard du programme**) peut aussi être utilisé pour lire une ligne dans un fichier, si vous lui précisez sa provenance.

Nous allons pouvoir maintenant appliquer ces quelques notions dans un ou deux petits programmes.

Les fichiers et la boucle **while**

Il a sûrement dû vous paraître surprenant que nous abordions l'interaction avec les fichiers dans un chapitre sur les boucles. En réalité, ça le serait vraiment dans n'importe quel autre langage que Perl. La différence avec les autres langages tient dans le fait que Perl a été pensé de telle manière à ce que la syntaxe de lecture des fichiers épouse parfaitement celle de la boucle **while**.

En effet, lorsque Perl lit un fichier, l'instruction `$ligne = <$fh>;` est systématiquement évaluée comme étant vraie, jusqu'à ce que le fichier soit entièrement lu. Ainsi, il est tout à fait naturel, en Perl, de lire un fichier complet, ligne par ligne, de la façon suivante :

Code : Perl

```
use strict;
use warnings;

my $filename = "fichier_test.txt";
open (my $fh, '<', $filename) or die "Erreur : impossible d'ouvrir
le fichier '$filename'";

while (my $line = <$fh>)
{
    print $line;
}
close $fh;
```

Exercices

Numérotation de lignes

Écrivez un programme en Perl qui ouvre un fichier, et l'affiche en numérotant ses lignes dans la console.

Par exemple, avec ce fichier texte :

Citation : lorem ipsum

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla porta adipiscing purus vitae pretium. Fusce leo libero, placerat eu lacinia non, malesuada eu eros. Nullam mi leo, adipiscing nec sollicitudin eu, varius at velit. Donec molestie lobortis accumsan. Vestibulum iaculis, ligula sit amet aliquet pretium, elit elit eleifend metus, non molestie turpis turpis ac nibh. Donec dictum nunc vel ligula commodo ac pellentesque odio blandit. Aliquam et diam eu augue tempor rutrum. Integer eleifend lorem et arcu pellentesque interdum. Integer ac mi leo. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Curabitur nunc elit, pellentesque sollicitudin congue at, gravida ac nulla. Vestibulum faucibus risus in turpis tristique adipiscing. Vestibulum tempor, mi eu imperdiet ornare, lectus sapien dapibus diam, eget facilisis mi enim ac sapien. Fusce vel enim justo. Etiam suscipit iaculis commodo.

Le résultat produit devrait ressembler à ceci :

Code : Console

```
arnaud@laptop(perl)% perl numlignes.pl
1      Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla porta adipis
2      purus vitae pretium. Fusce leo libero, placerat eu lacinia non, malesuada e
```

```
3      eros. Nullam mi leo, adipiscing nec sollicitudin eu, varius at velit. Donec
4      molestie lobortis accumsan. Vestibulum iaculis, ligula sit amet aliquet
5      pretium, elit elit eleifend metus, non molestie turpis turpis ac nibh. Done
6      dictum nunc vel ligula commodo ac pellentesque odio blandit. Aliquam et dia
7      augue tempor rutrum. Integer eleifend lorem et arcu pellentesque interdum.
8      Integer ac mi leo. Cum sociis natoque penatibus et magnis dis parturient
9      montes, nascetur ridiculus mus. Curabitur nunc elit, pellentesque sollicitu
10     congue at, gravida ac nulla. Vestibulum faucibus risus in turpis tristique
11     adipiscing. Vestibulum tempor, mi eu imperdiet ornare, lectus sapien dapibu
12     diam, eget facilisis mi enim ac sapien. Fusce vel enim justo. Etiam suscipi
13     iaculis commodo.
```

Secret (cliquez pour afficher)

Code : Perl

```
use strict;
use warnings;

my $filename = "/home/arnaud/prg/perl/lorem_ipsum.txt";

open (my $fh, '<', $filename)
    or die "Impossible d'ouvrir le fichier '$filename' en
lecture";

my $compteur = 1;

while (my $ligne = <$fh>)
{
    print "$compteur\t$ligne";
    ++$compteur;
}

close $fh;
```

Copier un fichier

Écrivez un programme qui recopie un fichier texte fichier_test1.txt dans un nouveau fichier appelé fichier_test2.txt.

Secret (cliquez pour afficher)

Code : Perl

```
use strict;
use warnings;

my $fichier_src = "fichier_test1.txt";
my $fichier_dst = "fichier_test2.txt";

open (my $fh_src, '<', $fichier_src)
    or die "impossible d'ouvrir le fichier '$fichier_src' en
lecture";

open (my $fh_dst, '>', $fichier_dst)
    or die "impossible d'ouvrir le fichier '$fichier_dst' en
écriture";
```

```
while (my $ligne = <$fh_src>)
{
    print $fh_dst $ligne;
}

close $fh_src;
close $fh_dst;
```

La variable implicite \$_

Il est temps maintenant de faire connaissance avec votre premier mécanisme magique en Perl. Essayez de deviner ce que fait le programme suivant :

Code : Perl

```
use strict;
use warnings;

my $filename = "fichier_test.txt";
open (my $fh, '<', $filename)
    or die "Erreur : impossible d'ouvrir le fichier '$filename'";

# Vous prendrez bien une petite tasse de magie noire ?
while(<$fh>)
{
    print;
}

close $fh;
```

Cette boucle bizarre, qui semble n'utiliser aucune variable, lit le contenu d'un fichier et l'affiche dans la console. Elle repose sur le mécanisme de la variable implicite `$_` que l'on appelle et prononce "it" (comme dans rhino-pharyngite).

En Perl, chaque fois qu'une instruction est exécutée, sa valeur de retour est assignée à cette variable spéciale. C'est ce qui se passe chaque fois que l'instruction `<$fh>` est exécutée dans le prédicat de la boucle `while`.

Lorsque l'on appelle ensuite la fonction `print` sans argument (mais ce n'est pas réservé à `print`, ça pourrait aussi être `chomp` ou une autre fonction...), Perl, ne trouvant pas d'argument, va utiliser par défaut le contenu de cette variable.

Pour illustrer ce comportement, vous pouvez essayer ceci :

Code : Perl

```
use strict;
use warnings;

print;
```

Étant donné que la seule instruction de ce programme est un `print` sans argument (sous-entendu, qu'aucune instruction n'a été évaluée avant), perl va vous donner le warning suivant :

Code : Console

```
Use of uninitialized value $_ in print at test.pl line 4.
```

Comme vous le voyez, c'est bien cette variable `$_` qui a été passée par défaut à **print**.

En dehors de cela, cette variable se comporte exactement comme toutes les autres. Vous pouvez donc lui assigner la valeur de votre choix, par exemple. Ce comportement permet simplement d'écrire **beaucoup moins de code** pour réaliser une même opération.



La vérité sur l'opérateur <>

Depuis le début de ce cours, et jusqu'à ce que nous voyions ensemble que cet opérateur pouvait être utilisé avec des variables, on vous a dit que l'instruction `<>;` servait à lire sur l'entrée standard du programme. En réalité, *ce n'est pas tout à fait exact*. En effet, il est aussi possible de lire dans des fichiers au moyen de cette instruction, sans même s'embêter à ouvrir ceux-ci nous-mêmes.

Pour cela, il suffit de passer le chemin d'un fichier en argument de votre script.

Démonstration avec le code suivant, qui reproduit, du coup, exactement le comportement de l'utilitaire `cat` sous unix :

Code : Perl

```
use strict;
use warnings;

while (<>)
{
    print;
}

# Ajoutez cette ligne sous Windows, au lieu de <>;
<STDIN>;
```

Pour passer un fichier texte à ce programme, il suffit de l'appeler comme ceci en ligne de commande :

Code : Console

```
arnaud@laptop(perl)% perl cat.pl lorem_ipsum.txt
```

Vous pouvez bien entendu lui en passer plusieurs : le programme affichera tous ces fichiers dans la console.



Si vous êtes sous Windows, alternativement, il vous suffit de **glisser-déposer** un fichier texte sur l'icône du programme.

Nous apprendrons à gérer plus finement ces arguments dans le prochain chapitre.

Pour l'heure, vous remarquerez que sous Windows, nous avons utilisé l'instruction `<STDIN>;` au lieu de `<>;` afin de marquer la traditionnelle pause en fin de programme. Ceci permet de spécifier explicitement, au moyen du filehandle spécial `STDIN`, que vous voulez attendre une entrée utilisateur sur l'entrée standard du programme (plutôt que d'éventuellement lire dans un fichier). De même, nous aurons l'occasion de manipuler d'autres filehandles spéciaux de ce type en temps voulu. 😊

D'autres types de boucles

Maintenant que vous vous êtes familiarisé avec la boucle `while` tout au long de ce chapitre, il ne nous reste plus qu'à passer rapidement en revue les autres constructions permettant de créer des boucles, qui ne sont rien de plus que d'autres façons d'exprimer la même chose. *There Is More Than One Way to Do It* !

La boucle **until**

La boucle `until` n'est rien d'autre que le contraire du `while`. En effet, au lieu d'exécuter une action en boucle *tant que* le prédicat est vérifié, celle-ci sera effectuée *jusqu'à ce qu'il* le soit. Voici le jeu du plus ou moins implémenté avec une boucle `until` :

Code : Perl

```
use strict;
use warnings;

my $nb_mystere = int (rand 101);
my $essai = -1;

print "Devinez le nombre mystère.\n";

# Répéter JUSQU'À CE QUE le joueur ait trouvé le nombre
until ($essai == $nb_mystere)
{
    print '> ';
    $essai = <>;

    if ($essai < $nb_mystere)
    {
        print "C'est plus.\n";
    }
    elsif ($essai > $nb_mystere)
    {
        print "C'est moins.\n";
    }
}

print "Gagné !";
```

La boucle `do ... while`

Une autre boucle qui découle directement de `while` est la boucle `do ... while`. Celle-ci permet d'exécuter le code de la boucle une première fois d'office, et de le répéter si le prédicat est vérifié. En somme, cela permet d'évaluer le prédicat à la fin du bloc, plutôt qu'au début, et donc de s'affranchir de certaines initialisations qui ne sont pas très naturelles :

Code : Perl

```
use strict;
use warnings;

my $nb_mystere = int (rand 101);
my $essai; # Pas besoin d'initialiser cette variable
           # avec une valeur arbitraire.

print "Devinez le nombre mystère.\n";

do
{
    print '> ';
    $essai = <>;

    if ($essai < $nb_mystere)
    {
        print "C'est plus.\n";
    }
    elsif ($essai > $nb_mystere)
    {
        print "C'est moins.\n";
    }
} while ($essai != $nb_mystere);

print "Gagné !";
```

La boucle **for**

Cette boucle est un petit peu plus compliquée que les précédentes. En effet, au lieu de reposer sur un prédicat, celle-ci fonctionne avec trois instructions et permet de gérer facilement un compteur.

```
for (INITIALISATION; PRÉDICAT; MISE_À_JOUR)
```

Où :

- INITIALISATION est une instruction permettant d'initialiser son compteur,
- PRÉDICAT est le prédicat à vérifier, exactement comme dans une boucle **while**,
- MISE_À_JOUR est une instruction exécutée à la fin de chaque passage en boucle, et que l'on utilise classiquement pour incrémenter son compteur.

Voici par exemple un programme tout simple qui compte de 0 jusqu'à 30 au moyen d'une boucle **for** :

Code : Perl

```
use strict;  
use warnings;  
  
# Le compteur est initialisé à 0  
# On compte jusqu'à ce qu'il soit égal à 30  
# Pour le mettre à jour, on incrémente sa valeur de 1  
for (my $compteur = 0; $compteur <= 30; $compteur++)  
{  
    print "$compteur\n";  
}
```

Cette boucle **for** peut vous sembler déroutante au premier abord. Pour vous y retrouver, gardez en tête que ce n'est qu'un **while** déguisé, et remarquez que ces deux constructions sont équivalentes tant que l'initialisation et la mise à jour tiennent en une seule instruction chacune :

Code : Perl

```
# Ceci :  
  
INITIALISATION;  
while (PRÉDICAT)  
{  
    MISE_A_JOUR;  
    # Corps de la boucle  
}  
  
# ... est équivalent à cela :  
  
for (INITIALISATION; PRÉDICAT; MISE_A_JOUR)  
{  
    # Corps de la boucle  
}
```

La syntaxe en ligne

Enfin, concluons ce chapitre en abordant rapidement la boucle **while** en ligne. À l'image de **if** et **unless**, il est possible de placer un **while** derrière une instruction afin de répéter celle-ci.

L'exemple qui vous parlera le plus sera peut-être ce programme qui tient en une seule ligne, et qui reproduit à l'identique le comportement de l'utilitaire `cat` :

Code : Perl

```
print while (<>) ;
```

Il est évident qu'autant de magie noire dans un seul programme est un petit peu hors de votre portée pour le moment, mais il est tout de même utile de vous montrer que ce genre de choses existe, afin que vous vous y habituiez, et que ces raccourcis finissent par vous venir naturellement à l'esprit lorsque vous aurez progressé. 😊

Exercice

À titre d'exercice, pour conclure ce chapitre, écrivez un programme qui écrit la table de multiplication du nombre entré par l'utilisateur, de 0 à 100, dans un fichier.

Par exemple, si l'utilisateur entre "7", votre programme devra écrire dans le fichier :

Code : autre

```
0 x 7 = 0
1 x 7 = 7
2 x 7 = 14

[ ... ]

99 x 7 = 693
100 x 7 = 700
```

Écrivez ce code quatre fois en utilisant respectivement une boucle **while**, une boucle **do ... while**, une boucle **until** et une boucle **for**.

Selon vous, quel type de boucle est le plus adapté à ce programme ? Pourquoi ?



Et la correction, où est-elle ?

Vous ne l'aurez pas cette fois ! Si votre fichier final contient la bonne sortie, c'est que votre programme fonctionne et que vous avez compris ce chapitre. Sinon, cherchez ce qui ne fonctionne pas et corrigez-le vous-même. Vous avez toutes les données et les explications nécessaires pour réussir cet exercice dans ce chapitre, et il est très important que vous soyez capables, dès à présent, de réussir un tel exercice sans reposer sur une correction qui ne sera plus là *dans la vraie vie*. Allez, au boulot ! 😊

Voilà un beau chapitre d'abattu !

Vous avez découvert beaucoup de notions nouvelles dans ce chapitre. Celles-ci vous permettront de créer des programmes de plus en plus complexes, et de commencer à interagir le contenu de votre disque dur. Prenez bien le temps de lire, voire relire ce chapitre, afin de bien digérer ce que nous venons de voir, car nous en aurons besoin dans toute la suite du cours !

Vous avez maintenant une idée ce qu'est la programmation en Perl. Vous avez aussi assez de connaissances pour écrire quelques scripts de base. Avant de passer à la suite, assurez-vous de bien avoir assimilé les chapitres précédents et, surtout, pratiquez sans relâche !

Ce tutoriel est encore en cours d'écriture. N'hésitez pas à laisser des commentaires sur [le topic dédié](#), afin de nous aider à le perfectionner.

Remerciements

Merci à [Talus](#) pour ses screenshots sous Windows, ainsi qu'à toute l'équipe de validation du Site du Zéro pour leur relecture attentive.