

Outils Perl pour le système



Sylvain Lhullier
contact@formation-perl.fr
<https://formation-perl.fr/>

La reproduction et/ou diffusion de ce document, même partielle, quel que soit le support, numérique ou non, est strictement interdite sans autorisation écrite des ayants droit.



Plan du cours

Bonnes pratiques	Courriels
Appels système	* réception
* système de fichiers	* envoi
* processus et IPC	* analyse
Formats mémoire et binaires	Gestion des paramètres
Gestion du temps	Filtres courts avec Perl
Sockets	Lancer une commande
Threads	Fonction <code>eval</code>



Bonnes pratiques en Perl

Toujours coder comme si le gars qui va finir par maintenir votre code est un psychopathe violent qui sait où vous habitez. – Rick Osborne

Des idées, des pistes. Pas des règles immuables.

Référence : Damian Conway – O'Reilly
Perl best practices / De l'art de programmer en Perl

Quick reference guide (2 pages) :
http://www.squirrel.nl/pub/PBP_refguide-1.02.00.pdf



Bonnes pratiques : Agencement du code

- Ouvrez les accolades et les parenthèses en fin de ligne
- Mettez un point-virgule après chaque instruction
- Mettez une virgule après chaque entrée de liste
- Codez en 80 caractères de large
- Indentez avec 3 ou 4 espaces, pas de tabulation
- Ne mettez pas 2 instructions sur la même ligne
- Alignez verticalement les éléments correspondants
- Coupez les longues instructions avant un opérateur



Bonnes pratiques : Conventions de nommage

- Nommez les tableaux au pluriel, les hash au singulier
- Nommez les variables stockant des références avec le suffixe `_ref`
- Utilisez l'underscore `_` pour séparer les mots
- Abrégez les noms lors que ceux-ci sont totalement non ambigus
- Préfixez avec l'underscore `_` les fonctions et variables à usage interne



Bonnes pratiques : Valeurs et expressions

- Utilisez `"` uniquement pour interpoler
- Utilisez l'underscore pour les nombres longs `1_000_000`
- Gardez la double-flèche `=>` pour les couples clef-valeur
- N'utilisez pas la virgule pour séparer les instructions
- N'utilisez pas les références symboliques



Bonnes pratiques : Variables

- Utilisez des noms lexicaux (mots lisibles)
- N'utilisez ni mots-clefs ni fonction du langage comme nom
- Ne modifiez pas `$_`
- Utilisez les indices négatifs pour accéder à la fin d'un tableau
- Si vous devez modifier une variable ponctuation, localisez la

```
{ local $/ = undef; $content = <$fd>; }
```
- Si vous devez modifier une variable d'un module, localisez la



Bonnes pratiques : Structures de contrôle

- N'utilisez ni `unless` ni `until`
- N'utilisez pas les `if` `for` `while` postfix
- Utilisez une variable déclarée `my` comme variable de boucle `for`
- Utilisez `foreach` pour les parcours et `map` pour les transformations
- Utilisez `for` et `redo` plutôt qu'un `while` irrégulièrement compté



Bonnes pratiques : Documentation

- Distinguez doc utilisateur et technique
- Utilisez POD dans le code pour la doc utilisateur
- Décrivez vos algorithmes
- Documentez tous ce sur quoi vous avez passé du temps
- Imaginez devoir ré-écrire si vous ne commentez pas



Bonnes pratiques : Fonctions du langage

- Utilisez **reverse** pour inverser une liste
- N'utilisez pas **eval** sur une chaîne
- Utilisez **glob** plutôt que `<...>`
- Utilisez un block d'instruction avec **map** **grep** **sort**
- Ne modifiez pas les valeurs dans un **sort**
- Découpez avec **unpack** si taille fixe
- Découpez avec **split** si séparateur simple
- Découpez avec **Text::CSV_XS** si séparateur complexe



Bonnes pratiques : Vos fonctions

- N'appellez pas les fonctions avec leur sigil **&**
- N'utilisez ni mots-clefs ni fonction du langage comme nom
- Recopiez **@_** en première instruction
- N'utilisez ni **@_** ni **\$_[..]** ensuite
- Utilisez une hash de paramètres si la fonction a beaucoup de paramètres
- Testez **defined** ou **exists** sur les paramètres
- N'utilisez pas les prototypes de Perl 5
- Renvoyez vos valeur avec un **return** explicite



Bonnes pratiques : hash de paramètres

```
tache( 'Paul', 21, undef, 7, undef, 42 );  
tache( prenom=>'Paul', age=>21,  
       niveau=>7, pointure=>42 );  
  
sub tache {  
    my %param = @_;  
    $param{prenom}    $param{age}  
    ...  
}
```



Bonnes pratiques : Entrées/sorties

- N'utilisez ni mots-clefs ni fonction du langage comme nom de descripteur
- Utilisez `open` avec 3 arguments et non 2
- Fermez les descripteurs explicitement au plus tôt
- Testez vos `open print close`
- Utilisez `while(<>)` et non `for(<>)`
- Pour `print`, mettez entre accolades le descripteur
`print {$fd} ''`



Bonnes pratiques : Modules

- Imaginez l'interface du module avant de coder
- Une variable ne doit pas faire partie de l'interface du module
- Code dupliqué \Rightarrow fonction / fonction dupliée \Rightarrow module
- Indiquez un numéro de version dans vos modules
- Exportez judicieusement et si possible sur demande
- Ne ré-inventez pas la roue, utilisez CPAN



Bonnes pratiques : Objet

- Ne faites pas de l'objet par défaut, ce doit être un choix
- Encapsulez totalement vos objets
- Utilisez toujours le même nom pour le constructeur
- Appliquez aux méthodes les règles des fonctions
- Écrivez des accesseurs
- Utilisez la surcharge d'opérateur avec parcimonie
- N'utilisez pas `AUTOLOAD`



Bonnes pratiques : Tests et débogage

- Utilisez toujours `use strict;`
- Utilisez toujours `use warnings;`
- Ne croyez pas qu'une compilation sans erreur donne un programme correct
- Écrivez les tests avant la fonction (TDD test driven development)
- Faites vos tests avec `Test::More`
- Faites vos suites de tests avec `Test::Harness`
(harnachement attelage)



Bonnes pratiques : Divers

- Utilisez un système de gestion de version
- Utilisez une syntaxe de configuration simple
- Ne tentez pas d'optimiser sans mesurer
- Pensez aux caches (pensez à en mesurer l'efficacité)

Critiques de votre code :

- module `Perl::Critic`
- commande `perlcritic`

Des bonnes idées, des pistes. Pas des règles immuables.



Appels système

Comme le C, Perl dispose de tous les appels système.
Les interfaces peuvent changer un peu.
Portabilité sous windows, etc.

Appel système : routine en interaction avec le noyau.
Ne pas confondre avec la commande `system('')` !

`perldoc perlfunc`



Appels système : gestion du système de fichiers

- `chmod mode, fichiers`
Changement des permissions des fichiers
`chmod 0644, @filenames;`
- `umask mode`
Changement de la valeur d'umask
`$oldumask = umask 0022;`
- `chown $uid, $gid, @filenames;`
Changement de propriétaires et/ou de groupe.



Appels système : gestion du système de fichiers

- `rename $oldname, $newname;`
Déplace un fichier/dossier (dans le même FS)
`File::Copy move` - cross-FS
`File::Rename rename` - renommage multifichier
- `unlink @filenames;`
Supprime les fichiers
`File::Remove remove` - suppression récursive



Appels système : gestion du système de fichiers

- `chdir '/new/path';` - Changement de répertoire courant
- `mkdir $newdir(, $mask);` - Création d'un nouveau répertoire
`File::Path mkpath` - création en profondeur
- `rmdir $dir;` - Suppression d'un répertoire vide
- `use POSIX qw(getcwd);`
`$pwd = getcwd;` - Renvoie le répertoire courant
Le module `Cwd` gère les problématiques de liens symboliques



Appels système : gestion du système de fichiers

Manipulation fine du contenu des répertoires

- `opendir` - ouverture
- `readdir` - lecture
- `closedir` - fermeture
- `rewinddir` - rembobinage
- `telldir` - position actuelle
- `seekdir` - déplacement à une position

Rappel : `@f = glob('/path/*.pl');`



Appels système : gestion du système de fichiers

- `link $existingfile, $newlink`
Création d'un lien physique (même numéro d'inode)
- `symlink $existingfile, $newsymlink`
Création d'un lien symbolique



Appels système : gestion du système de fichiers

`stat` - récupération d'information sur un fichier

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

0 `dev` - numéro de device du filesystem

1 `ino` - numéro d'inode

2 `mode` - mode du fichier (type et permissions)

3 `nlink` - nombre de liens physiques

4 `uid` - numéro du propriétaire

5 `gid` - numéro du groupe

6 `rdev` - identifiant du device (fichier spécial)

7 `size` - taille en octets



Appels système : gestion du système de fichiers

8 `atime` - dernier accès (secondes depuis 1970)
9 `mtime` - dernière modification (secondes depuis 1970)
10 `ctime` - dernière modification de l'inode (secondes depuis 1970)
11 `blksize` - taille des blocks du FS
12 `blocks` - nombre de blocks utilisés

```
(undef,undef,undef,undef,$uid,$gid,undef,$size) = stat($filename);  
$mode = (stat($filename))[2];  
($mode,$uid,$gid) = (stat($filename))[2,4,5];
```

`lstat` : information sur le lien et non le fichier pointé



Appels système : dossier/fichier temporaire

```
use File::Temp qw/tempfile tempdir/; #Interface objet disponible
```

Création de fichier temporaire :

```
$fd = tempfile(); # Descripteur ouvert  
($fd, $filename) = tempfile(); # Connaître le nom  
$template = 'tempXXXX'; # Modèle de nom de fichier  
($fd, $filename) = tempfile( $template, SUFFIX => '.dat');  
($fd, $filename) = tempfile( $template, DIR => $dir);
```

Création de dossier temporaire (et utilisation par `tempfile`) :

```
$dir = tempdir(); # Dossier créé  
$dir = tempdir( CLEANUP => 1 ); # Suppression automatique  
($fd, $filename) = tempfile( DIR => $dir );
```



Appels système : gestion des utilisateurs/groupes

Consultation de la base de utilisateurs / groupes
(lecture des fichiers `/etc/passwd` et `/etc/group` et/ou base PAM) :

```
• ($login,$passwd,$uid,$gid,$quota,$comment,$gcos,$dir,  
  $shell,$expire) = getpwuid($uid);
```

Informations sur un utilisateur à partir de son numéro

```
• ($name,$passwd,$gid,$members) = getgrgid($gid);
```

Informations sur un groupe à partir de son numéro



Appels système : gestion des processus

- `$$` `$PID` `$PROCESS_ID` - numéro de PID
- `getppid` - numéro de PID du père
- `fork` - création d'un processus fils
- `exec` - exécute un programme (écrasement de code)
- `pipe` - création d'un tuyau
- `wait` - attendre la fin d'un processus quelconque
- `waitpid` - attendre la fin d'un processus donné
- `kill` - envoyer un signal (lire la suite)
- `%SIG` - gestion des signaux (lire la suite)



Appels système : gestion des processus

```
pipe( my $readhandle, my $writehandle ) or die("pipe: $!");
my $pid;
if( ($pid = fork()) == -1 ) { die("fork: $!"); }
if( $pid != 0 ) { #Père
    close($readhandle);
    $writehandle->autoflush(1); #use IO::Handle;
    while( .. ) {
        print $writehandle ".....\n";
    }
} else { #Fils
    close($writehandle);
    while( defined( my $ligne = <$readhandle> ) ) {
    }
}
```



Appels système IPC : gestion des signaux

Table de hachage %SIG Clefs : noms des signaux
Valeurs : références (de code) de handlers pour ces signaux

```
my $get_signal = 0;
$SIG{INT} = $SIG{QUIT} = sub { $get_signal++ };
while ( not $get_signal ) {
    # traitement long
}
$SIG{INT} = \&fonction_de_traitement;
```

Le pragma sigtrap permet de gérer les signaux également

```
use sigtrap 'handler' => \&my_handler, 'INT';
```



Appels système IPC : gestion des signaux

Pour lancer un signal :

```
kill 'INT', $child1, $child2;
kill 'SIGINT', @childs;
```

```
use Proc::Killall;
killall('HUP', 'xterm'); # SIGHUP all xterms
```

```
⇒ man 7 signal
⇒ perldoc perlvar section %SIG
⇒ perldoc sigtrap
⇒ perldoc -f kill
```



Appels système : gestion des IPC

- File de messages
msgctl msgget msgrcv msgsnd
- Mémoire partagée
shmctl shmget shmread shmwrite
- Sémaphore
semctl semget semop



Appels système : gestion du système de fichiers

- `fcntl`
Manipulation des descripteurs de fichier
- `ioctl`
Modifie le comportement des périphériques sous-jacents des fichiers spéciaux.



Appels système : gestion du système de fichiers

- `utime $atime, $mtime, @filenames;`
Change les dates des fichiers (secondes depuis 1970)
- `binmode $fd;`
Passe les écritures/lectures à venir en mode binaire.
Pas de transformation dues au texte : `\n` windows, etc



Fonction utile : `rand`

La fonction `rand` renvoie un nombre aléatoire entre 0 et son premier paramètre (1 par défaut)

`rand()` renvoie un nombre décimal entre 0 et 1 (1 exclu)

`rand(256)` renvoie un nombre décimal entre 0 et 255

`int(rand(256))` renvoie un nombre entier entre 0 et 255

La fonction `srand` est appelée automatiquement par `rand` si cela n'a jamais été fait.



Fonctions utiles : attendre

- `sleep($seconds);`
- `use Time::HiRes qw(usleep);`
`usleep($microseconds);`
- `use Time::HiRes qw(nanosleep);`
`nanosleep($nanoseconds);`



Formats mémoire et binaires

Conversion de données : format Perl <=> format binaire

- `pack` format Perl => format binaire
- `unpack` format binaire => format Perl

```
$binaryint = pack('i', 125);  
$binarydouble = pack('D', 3.14159);  
$valeur = unpack('s', $binaryshort);
```



Formats mémoire et binaires

Code de formats :

- `'c'` char - `'C'` unsigned char
- `'s'` short - `'S'` unsigned short
- `'i'` int - `'I'` unsigned int
- `'F'` float
- `'D'` double
- etc

`perldoc -f pack`



Accès en parallèle à des flux avec select

L'appel système `select` permet une attente non-coûteuse sur plusieurs flux :

```
select( RBITS, WBITS, EBITS, TIMEOUT )
```

- `RBITS` : flux observés afin d'y lire
- `WBITS` : flux observés afin d'y écrire
- `EBITS` : flux observés pour des événements exceptionnels
- `TIMEOUT` : délai maximum d'attente (undef pour infini)



Accès en parallèle à des flux avec select

Il faut construire les champs de bits :

```
my ($rin, $win, $ein);  
$rin = $win = $ein = '';  
# Descripteurs en lecture  
vec($rin, fileno(STDIN), 1) = 1;  
vec($rin, fileno($fd1), 1) = 1;  
# Descripteurs en écriture  
vec($win, fileno(STDOUT), 1) = 1;  
vec($win, fileno($fd2), 1) = 1;  
# Descripteurs pour événements  
$ein = $rin | $win;
```



Accès en parallèle à des flux avec select

Puis appel à select :

```
# Appel classique :
($nfound,$timeleft) =
    select($rout=$rin, $wout=$win, $eout=$ein, $timeout);
```

- \$nfound : nombre de flux en attente (utilisables)
- \$timeleft : temps restant
- \$rout, \$wout et \$eout : vecteurs indiquant les flux en attente



Accès en parallèle à des flux avec select

Autres appels à select :

```
# Appel sans attente maximale :
($nfound,$timeleft) =
    select($rout=$rin, $wout=$win, $eout=$ein, undef);

# Appel en contexte scalaire :
$nfound = select( ... );

# Attente de 300 millisecondes :
select(undef, undef, undef, 0.3);

perldoc -f select
```



Accès en parallèle à des flux avec IO::Select

```
use IO::Select;
$s = IO::Select->new();
$s->add(*STDIN);
$s->add($some_handle);
@ready = $s->can_read($timeout);
# $s->can_write( ... );
# $s->has_exception( ... );

@ready = IO::Select->new(@handles)->can_read($timeout);

perldoc IO::Select
```



Gestion d'événements multiples avec AnyEvent

Appel de callback (cb) selon l'événement

```
use AnyEvent;
# Attente de pouvoir lire dans un flux
AnyEvent->io( fh=>$fd, poll=>'r', cb=>sub{ ... } );

# Timer unique ou plusieurs fois
AnyEvent->timer( after=>$seconds, cb=>sub { ... } );
AnyEvent->timer( after=>$sec1, interval=>$sec2, cb=>... );

# Signal POSIX
AnyEvent->signal( signal=>'TERM', cb=>sub{ ... } );

# Fin de sous-processus
AnyEvent->child( pid=>$pid, cb=>sub{ ... } );
```



Gestion du temps

Système de référence :

- Unité : seconde
- *Epoch* : 1er janvier 1970 à 00h00 UTC (0 UNIX)
UTC (Universal Time Coordinated). Ne pas confondre avec
GMT (Greenwich Mean Time) : heure observée à Greenwich
(vitesse variable de la Terre sur son orbite elliptique)

Variantes :

- Décalage depuis *Epoch*
- Fuseau horaire local



Gestion du temps : secondes depuis *Epoch*

`time()`

Renvoie le nombre actuel de secondes depuis *Epoch*

`localtime()` convertit un nombre de seconde depuis *Epoch*
en une représentation locale

- En contexte de chaîne : chaîne de `ctime`
`$now_string = localtime(); # 'Mon Oct 31 13:25:46 2042'`
- En contexte de liste :
`($sec,$min,$hour,$mday,$mon,$year,$yday,$isdst) =
localtime();`



Gestion du temps : valeurs renvoyées par `localtime`

0 `sec` - nombre de secondes (de 0 à 59, voire 60)

1 `min` - nombre de minutes

2 `hour` - nombre d'heures

3 `mday` - numéro de jour dans le mois

4 `mon` - mois (0=janvier, 11=décembre)

5 `year` - année-1900 (142 pour 2042)

6 `wday` - jour de la semaine (0=dimanche, 6=samedi)

7 `yday` - jour de l'année (de 0 à 364 ou 365 / 0=1er janv)

8 `isdst` - booléen indiquant que l'on est ou non en période
de changement d'heure d'été / heure d'hiver

```
($sec,$min,$hour,$mday,$mon,$year,$yday) = localtime();
```

```
$year += 1900; @d = qw(Dim Lun Mar Mer Jeu Ven Sam);
```

```
@m = qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);
```

```
print "$hour:$min:$sec $d[$yday] $mday $m[$mon] $year\n";
```



Gestion du temps : paramètre de `localtime`

`localtime()` peut prendre un paramètre : un nombre de seconde
depuis *Epoch*, par défaut : `time()`

```
localtime(time()-60*60*24*2); Il y a deux jours
```

```
localtime((stat($filename))[9]); Date de modif. du fichier
```

Agir sur le fuseau horaire : `$ENV{TZ}` (lire plus loin)

`gmtime()` fait la même chose mais en temps UTC

Attention au nom : il s'agit bien de UTC

Attention : un seul *t* !



Gestion du temps : formatage avec strftime

`strftime()` permet de convertir les dates (string format time)
`perldoc POSIX` `man strftime`

`strftime(format, sec, min, hour, mday, mon, year)`

`sec` - nombre de secondes de 0 à 59

`min` - nombre de minutes

`hour` - nombre d'heures

`mday` - numéro de jour dans le mois

`mon` - mois (0=janvier, 11=décembre)

`year` - année-1900 (142 pour 2042)

(mêmes valeurs que celle renvoyées par `localtime`)



Gestion du temps : codes de format de strftime

Le format de `strftime()` est une chaîne de caractères
pouvant comporter :

`%A` - nom du jour de la semaine (suivant localisation)

`%B` - nom du mois (suivant localisation)

`%c` - représentation classique date&heure (suivant localisation)

`%d` - numéro du jour dans le mois (entre 01 et 31)

`%H` - heure (entre 00 et 23)

`%m` - numéro du mois (entre 01 et 12)

`%M` - minute (00 à 59)

`%s` - nombre de secondes écoulées depuis le Epoch

`%S` - seconde (00-59)

`%Y` - année (par exemple : 2042)

`%Z` - fuseau horaire

`%%` - caractère %



Gestion du temps : exemples de strftime

```
use POSIX qw(strftime);
```

```
$str = strftime('%A, %d %B %Y', 0, 0, 0, 15, 2, 142);  
print "$str\n"; #samedi, 15 mars 2042
```

```
$str = strftime('%A, %d %B %Y', localtime());  
print "$str\n"; #vendredi, 7 novembre 2042  
$str = strftime('%c', localtime());  
print "$str\n"; #ven 07 nov 2042 09:03:22 CET
```

```
$str = strftime('%d/%m/%Y %H:%M:%S', localtime($nbrSecEpoch));
```

```
$str=strftime('%d/%m/%Y %H:%M',gmtime(0)); #01/01/1970 00:00  
$str=strftime('%d/%m/%Y %H:%M',localtime(0));#01/01/1970 01:00
```



Gestion du temps : fuseau horaire

La fonction `localtime` est sensible à `$ENV{TZ}`

```
$ENV{TZ} = 'Europe/Paris';  
$str1 = strftime('%c', localtime($dateseconds));  
$ENV{TZ} = 'America/Montreal';  
$str2 = strftime('%c', localtime($dateseconds));
```

Possibilité de localiser l'affectation à `$ENV{TZ}` dans un bloc :

```
{  
    local $ENV{TZ} = 'Australia/Sydney';  
    $str3 = strftime('%c', localtime($dateseconds));  
}
```



Manipulation de dates avec Date::Manip

```
use Date::Manip; # 2 syntaxes : objet (vue ici) / fonctionnelle
# Analyser une date
$date = Date::Manip::Date->new;
$error = $date->parse('today');
$error = $date->parse('1st Thursday in June 1992');
$error = $date->parse('05/10/93');
$error = $date->parse('12:30 Dec 12th 1880');
$error = $date->parse('8:00pm December tenth');

my $sec = $date->printf('%s'); # Nombre de secondes depuis Epoch
my $string = $date->printf('%Y/%m/%d %H:%M:%S %Z');
# Codes de formats similaires à ceux de strftime
```

Pour plus d'infos `perldoc Date::Manip::Date`



Manipulation de dates avec Date::Manip

```
# Analyser un intervalle de temps
$delta = Date::Manip::Delta->new;
$error = $delta->parse('in 12 hours');
$error = $delta->parse('-0:1:30:0');
$error = $delta->parse('4 business days later');

# Calculer un intervalle de temps entre deux dates
$date1 = Date::Manip::Date->new;
$date2 = $date1->new_date();
$error = $date1->parse('Jan 30 2019 13:00 EST');
$error = $date2->parse('2/Mar/2019 15:30:00 +0500');
$delta = $date1->calc($date2);
# Objet de classe Date::Manip::Delta
```



Cervantes et Shakespeare sont morts à la même date mais pas le même jour ©Jean Forget

```
use DateTime::Calendar::Christian;
my $ws = DateTime::Calendar::Christian->new(
    year => 1616, month => 4, day => 23,
    reform_date => 'uk' );
my $mc = DateTime::Calendar::Christian->new(
    year => 1616, month => 4, day => 23,
    reform_date => 'italy' ); # même date que l'Espagne
my $dt_ws = DateTime->from_object(object => $ws); # pour avoir un overload
my $dt_mc = DateTime->from_object(object => $mc); # sur les comparaisons
my $comp = 'le même jour que';
if( $dt_mc < $dt_ws ) { $comp = 'avant' } else { $comp = 'après' }
print "Cervantes est mort $comp Shakespeare\n";
my $d = $dt_mc - $dt_ws;
print abs($d->delta_days), " jour(s) d'écart\n";
```

⇒ On apprend que Cervantes est mort 10 jours avant Shakespeare.

http://datetime.mongueurs.net/presentation/Dates_et_heures_en_Perl.html#modules_de_calendrier



Réseau : sockets

Deux interfaces principales :

- Appels système
`socket bind listen connect accept send recv`
- Interface objet
`perldoc IO::Socket`
`perldoc IO::Socket::INET`



Réseau : sockets serveur

```
use IO::Socket;
my $listen_socket = IO::Socket::INET->new(
    Proto=>'tcp', LocalPort=>2000, Listen=>5, Reuse=>1 )
    or die("$@");
# L'option Listen est obligatoire pour un serveur
while( my $accept_socket = $listen_socket->accept() ) {
    print "Nouveau client\n";
    $accept_socket->send( "Hello world\n" );
    close( $accept_socket );
}
```



Réseau : sockets client

```
use IO::Socket;
my $socket = IO::Socket::INET->new(
    Proto=>'tcp', PeerHost=>'localhost', PeerPort=>2000 )
    or die("$@");
while( defined( my $ligne = <$socket> ) ) {
    print "$ligne";
}
close( $socket );
```

Il existe aussi la méthode `recv` :

```
$socket->recv(my $buffer, 512);
```



Threads

Perl dispose de threads intégré depuis Perl 5.6.0
appelés threads interpréteur ou ithreads.
`perldoc threads` `perldoc perlthrtut`
Implémentation : généralement pthreads sous Linux

```
use threads;
```

Création : `threads->new(fonction, paramètres)`

- `$thr = threads->new(&fonction);`
 `sub fonction { print "Dans le thread\n"; }`
- `$thr = threads->new(&fonction, 'Param1', 'Param2');`
 `sub fonction { print "Dans le thread (@_)\n"; }`
- `$thr = threads->new(sub{print "Dans le thread\n";});`



Threads : gestion de la fin de vie

Gestion par le créateur du thread :

- `$thr->join();`
 Attendre qu'un thread termine et nettoyer la mémoire
 `sub fonction {; return $calcul; }`
 `$donnees_renvoyees = $thr->join();`
- `$thr->detach();`
 Ignorer la fin d'un thread (mémoire nettoyée automatiquement)



Threads

Pour qu'un thread rende explicitement le contrôle du processeur à un autre thread :

```
threads->yield();
```

(méthode de classe)

- cas de calculs très lourd et interface graphique
- gestion des threads ne pas supportant le multitâche préemptif



Threads : partage de données

Par défaut toutes les données sont propres au thread (même les variables globales)

Le package `threads::shared` permet d'indiquer lors de la déclaration de variables si elles sont partagées ou non.

Un scalaire, un tableau entier, une table de hachage entière peuvent être déclarés partagés :

```
my $var : shared = 2;
my @tab : shared = (4,5,6);
my %hash : shared = (Paul=>8);
```

L'atomicité de l'accès et modification d'une donnée est garantie.



Threads : partage de données

```
use threads;
use threads::shared; # Ordre important !

my $partage : shared = 1;
my $personnel = 1;
threads->new( sub{ $partage++; $personnel++ } )->join();

print "$partage\n";
# affiche 2 car $partage est partagée
print "$personnel\n";
# affiche 1 car $personnel n'est pas partagée
```



Threads : exclusion mutuelle

La fonction `lock()` pose un verrou sur une variable partagée.

```
my $verrou : shared = 0;
sub calcul {
    while( 1 ) {
        tache1();
        {
            lock($verrou); # Bloque jusqu'à obtenir le verrou
            tache2(); # Code avec exclusion mutuelle
        } # Verrou automatiquement rendu en fin de bloc
        tache3();
    }
}
my $t1 = threads->new(\&calcul);
my $t2 = threads->new(\&calcul);
$t1->join();
$t2->join();
```




Courriel

Un courriel (mail, email, etc) est constitué :

- d'une enveloppe (perdue à l'arrivée),
- d'en-têtes,
- d'un contenu.

Attachements : le contenu peut être composé de plusieurs entités.

SMTP : protocole d'envoi de courriels

POP3, IMAP : protocoles de réception de courriels

MIME : format des courriels



Exemple d'usage du module Net::POP3

```
#!/usr/bin/perl
use strict;
use warnings;
use Net::POP3;
my $pop = Net::POP3->new('host') or die("pop->new: $!");
$pop->login($username, $password) or die("pop->login: $!");
my $messages = $pop->list(); # hashref of msgId => size
foreach my $mesgId (sort {$a<=>$b} keys %$messages) {
    my $lineRef = $pop->get($mesgId); # tabref of lines
    print @$lineRef;
    $pop->delete($mesgId);
}
$pop->quit();
```



Courriel : envoi

Gestion complète du protocole SMTP : module Net::SMTP

- `my $smtp = Net::SMTP->new('mailhost');`
Connexion à un serveur
`Net::SMTP->new('mailhost', Port=>25, Timeout=>60);`
- `$smtp->mail('moi@ici.org');`
Adresse de l'expéditeur (MAIL FROM)
- `$smtp->to('lui@la-bas.org');`
Adresse du destinataire (RCPT TO)



Courriel : envoi

- `$smtp->data();`
Début des données (DATA)
- `$smtp->datasend('Blabla');`
Transmission des en-têtes et du contenu
- `$smtp->dataend();`
Fin des données (ligne avec point seul)
- `$smtp->quit();`
Ferme la connexion avec le serveur (QUIT).



Courriel : envoi

```
my $smtp = Net::SMTP->new('serveur', Timeout => 30);
$smtp->mail('moi@ici.org'); # Enveloppe
$smtp->to('lui@la-bas.org'); # Enveloppe
$smtp->data();
$smtp->datasend( "From: Moi <moi@ici.org>\n" );
$smtp->datasend( "To: Lui <lui@la-bas.org>\n" );
$smtp->datasend( "Subject: RVD demain\n" );
$smtp->datasend( "\n" );
$smtp->datasend( "C'est ok pour toi ?\n" );
$smtp->datasend( "\n" );
$smtp->datasend( "Moi.\n" );
$smtp->dataend();
$smtp->quit();
```



Courriel : TLS

Module Net::SMTP::TLS::ButMaintained (hérite de Net::SMTP)
car Net::SMTP::TLS non maintenu mais non abandonné.

```
my $smtp = Net::SMTP::TLS::ButMaintained->new(
    'mailhost',    Hello => 'machine.example.com',
    User => 'prenom.nom@example.com', Password => 's3cr3t' );
```

Puis utilisation des méthodes de Net::SMTP

```
$smtp->mail('moi@ici.org');
$smtp->to('lui@la-bas.org');
$smtp->data();
$smtp->datasend( ... );
$smtp->dataend();
$smtp->quit();
```



Courriel : envoi

Gestion du MIME : modules MIME:::
dont MIME::Lite qui génère des MIME::Entity.

Version simplissime :

```
use MIME::Lite;
$msg = MIME::Lite->new(
    From      => 'moi@ici.org',
    To        => 'lui@la-bas.org',
    Subject   => 'RVD demain',
    Data      => "C'est ok pour toi ?\n\nMoi.",
);
$msg->send();
```

Cette syntaxe nécessite un serveur de mail installé en local.



Courriel : envoi

- my \$mime = MIME::Lite->new(); - Construction

```
my $mime = MIME::Lite->new(
    From      => 'Moi <moi@ici.org>',
    To        => 'Lui <lui@la-bas.org>',
    Subject   => 'RVD demain',
    Type       => 'TEXT',
    Encoding  => 'quoted-printable',
    Data      => "C'est ok pour toi ?\n\nMoi.\n",
);
```



Courriel : envoi d'attachement

- `$mime->attach()`
Ajoute un attachement
- ```
$mime->attach(
 Type => 'image/jpeg',
 Encoding => 'base64',
 Path => '/path/to/file/badname.jpg',
 Filename => 'file.jpg'
);
```



## Courriel : méthodes de MIME::Entity

- `$mime->add('X-Mailer'=> 'monprog.pl V1.3');`  
Ajout/modification d'une en-tête
- `$field = $mime->get('X-Mailer');`  
Récupération de la valeur d'une en-tête
- `$mime->field_order('From', 'To', 'Subject');`  
Choix de l'ordre entre les en-têtes
- `$mime->sign(Path => '/home/moi/.signature');`  
Ajout d'une signature au contenu



## Courriel : envoi

- `$mime->as_string()`  
Renvoi les en-têtes+contenu sous forme de texte  
Utilisation : avec le module `Net::SMTP`  
ou `Net::SMTP::TLS::ButMaintained` :  
`$smtp->datasend( $mime->as_string() );`
- `$mime->send('smtp', 'mailhost.domain.tld');`  
Envoie ce courriel via le SMTP en question
- `MIME::Lite->send('smtp', 'mailhost.domain.tld');`  
Modifie les paramètres d'envoi pour tous les messages
- `$mime->send();` Envoie ce courriel en particulier



## Courriel : analyse

Analyse de courriels existants : module `MIME::Parser`  
Il génère des `MIME::Entity`

- `$parser = MIME::Parser->new();`  
Construction
- `$parser->output_dir('/tmp');`  
Spécifie le répertoire où seront stockés les attachements  
(par défaut : répertoire courant)
- `$parser->output_to_core(1);`  
Ne pas mettre les attachements sur disque mais en mémoire  
(attention aux gros attachements)



## Courriel : analyse

Analyse d'un courriel :

- `$mime = $parser->parse(\*STDIN);`  
`$mime = $parser->parse($fd);`  
Analyse d'un flux
- `$mime = $parser->parse_data($message);`  
Analyse d'un contenu présent en mémoire
- `$mime = $parser->parse_open($filename);`  
Analyse du contenu d'un fichier

`$mime` est un objet de classe `MIME::Entity`



## Courriel : gestion du type de contenu

Récupération du type de contenu : `$mime->get('Content-Type')`

Valeurs possibles : `text/plain` `text/html` `image/jpeg` etc  
(cf fichier `/etc/mime.types`)

Message comportant des attachements : `multipart/mixed`  
Notre `MIME::Entity` contient d'autres `MIME::Entity`.

`$mime->parts()`

Renvoie la liste des sous-parties (`MIME::Entity`)

`foreach my $submime ($mime->parts()) { }`



## Courriel : analyse du contenu

Obtenir le contenu d'un `MIME::Entity` :

`$lines = $mime->body();`

`$lines` est alors une référence vers un tableau des lignes.

```
foreach my $line (@$lines) { }
$full_text = join ' ', @$lines;
print @$lines;

foreach my $line (@{ $mime->body() }) {}
$full_text = join ' ', @{ $mime->body() };
print @{ $mime->body() };
```



## Courriel : gestion de l'encodage

Courriel = texte    Il faut encoder les données binaires

Récupération de l'encodage :

`$mime->get('Content-Transfer-Encoding')`

- 7bit : cas le plus simple
- 8bit : il faudra gérer le charset (`iso-8859-1`, `iso-8859-15`, etc.)
- quoted-printable  
Module `MIME::QuotedPrint` fonction `decode_qp`
- base64  
Module `MIME::Base64` fonction `decode_base64`



## Courriel : analyse

```
if($mime->get('Content-Type') =~ m:~multipart/mixed:) {
 foreach my $part ($mime->parts()) {
 my $content_type = $part->get('Content-Type');
 if($content_type =~ m:~text/:) {
 print @ { $part->body() };
 } elsif($content_type =~ m:~image/: and
 $part->get('Content-Transfer-Encoding') eq "base64\n") {
 open(my $fd,'>','image.jpg') or die("$!");
 binmode $fd;
 print $fd decode_base64(join ' ', @ { $part->body() });
 close $fd;
 }
 }
}
```



## Courriel : Mail::Box::Manager

Une boîte courriel est constituée de plusieurs courriels successifs.  
Le module Mail::Box::Manager permet d'en gérer les messages.

```
use Mail::Box::Manager;
my $mgr = Mail::Box::Manager->new;
my $folder = $mgr->open(folder => 'boite-mail.mbox');
foreach my $message ($folder->messages) {
 # Extraction de chaque courriel
 # (à traiter avec MIME::Parser ?)
 print $message->string;
}
```



## Gestion des paramètres avec Getopt::Long

Gestion des paramètres de la ligne de commande :

- flag, valeur numérique ou chaîne  
script.pl --verbose --nbr 42 --date now fichier
- valeurs multiples  
script.pl --rgbcolor 255 0 255 --coord 26.4 85.1
- paramètres condensés -l -a -c ou -lac
- syntaxe souple -d 32 -d32 --data=32 --data 32
- prise en compte de -- (fin des paramètres)  
script.pl --verbose --debug -- --nom-de-fichier.txt

Suppression de @ARGV des paramètres traités (lecture dans <>)



## Getopt::Long

```
use Getopt::Long;
my $file = 'data.txt';
my $size = 24;
my $debug;
my $r = GetOptions('size=i' => \$size, # nombre
 'file=s' => \$file, # chaîne
 'debug' => \$debug) # flag
 or die(usage());

Aliases :
GetOptions ('length|height=f' => \$length);
--length ou --height flottant
```



## Getopt::Long : type de paramètre

- flag  
`GetOptions('debug'=> \$debug);`
- entier  
`GetOptions('size=i'=> \$size);`
- flottant  
`GetOptions('length=f'=> \$length);`
- chaîne libre  
`GetOptions('file=s'=> \$file);`



## Getopt::Long : type de paramètre

- flag niable  
`GetOptions('debug!'=> \$debug);`  
Possibilité d'utiliser `--nodebug` et `--no-debug`
- paramètre incrémental  
`GetOptions('debug+'=> \$debug);`  
Chaque appel à `--debug` augmentera `$debug` de 1.
- paramètre avec ou sans valeur  
`GetOptions('file:s'=> \$file);` et non =  
Dans ce cas, `--file` peut de pas avoir de valeur, affectation de la chaîne vide dans la variable.



## Getopt::Long : type de paramètre

- valeurs multiples libres  
`GetOptions('path=s'=> \@path);`  
`--path chemin1 --path chemin2 --path chemin3`
- valeurs multiples contraintes  
`GetOptions('coord=f{2}'=> \@coord, 'rgbcolor=i{3}'=> \@color);`  
`--coord 26.4 85.1 --rgbcolor 255 0 255`
- valeurs multiples associatives  
`GetOptions('value=s'=> \%values);`  
`--value os=linux --value vendor=debian`



## Getopt::Long : callback

```
GetOptions('data' => sub{ });
GetOptions('data' => \&maFonction);

my $verbose = 0;
GetOptions ('verbose' => \$verbose,
 'quiet' => sub { $verbose = 0 });

sub f { print "f : @_ \n"; }
my $r = GetOptions('file=s' => \&f);
./prog.pl --file=data.txt
Arguments de la fonction f : file data.txt
```



## Filtres courts en Perl

Objectif : écrire des filtres en ligne de commande (one liner)  
aussi rapidement qu'en shell, mais plus puissants

```
perl ...options... fichier
who | perl ...options... | wc -l
```

Usage des options :

- **-e** : code en ligne de commande
- **-n** : lecture automatique
- **-p** : lecture et écriture automatiques
- **-i** : traitement «en ligne»
- **-M** : chargement d'un module



## Filtres courts en Perl : option -n

Option **-n** : lecture dans **\$\_** de chaque ligne de **<ARGV>**  
(fichiers passés en paramètres et/ou entrée standard).

Correspond à l'ajout des instructions suivantes :

```
LINE :
while (<>) {
 ... # votre programme ici
}
```



## Filtres courts en Perl : option -n

Exemple :

```
perl -ne 'print if m/regex/' fichier
exécute en fait :
```

```
LINE :
while (<>) {
 print if m/regex/
}
```

⇒ Affichage des lignes qui correspondent à une regexp Perl  
⇒ Commande **grep** avec les regexp de Perl



## Filtres courts en Perl : option -n

Affiche tous les logins du système

```
perl -ne 'm/^(.+?):/;print"$1\n"' /etc/passwd
```

Affiche la longueur de chaque ligne du fichier

```
perl -ne 'print length."\n"' fichier
```

Affiche uniquement les lignes numéros impairs (1ère, 3ème, etc)

```
perl -ne 'print if $.%2' fichier
```

Efface les fichiers non modifiés depuis 7 jours

```
find . -mtime +7 -print | perl -nle unlink
```



## Filtres courts en Perl : option -n

Compte le nombre de chiffres contenus dans le fichier  
`perl -ne '$_+=y/0-9//;END{print"$n\n"}' fichier`

Compte le nombre de «mots» contenus dans le fichier  
`perl -ne '@w=split/\W+/;$w+=@w;END{print"$w\n"}' fichier`

Supprime les lignes en doublon (mieux qu'`uniq`)  
`perl -ne 'print unless $cpt{$_}++' fichier`

Affiche les lignes du fichier par ordre décroissant d'occurrence  
`perl -ne '$_=$_++;END{print sort {$h{$b}<=>$h{$a}} keys%h}' f`



## Filtres courts en Perl : option -p

Option `-p` : lecture dans `$_` de chaque ligne de `<ARGV>`  
puis **affichage** de `$_` (qu'il faut donc modifier)

Correspond à l'ajout des instructions suivantes :

```
LINE:
while (<>) {
 ... # votre programme ici
} continue {
 print or die "-p destination: $_\n";
}
```



## Filtres courts en Perl : option -p

Numérote les lignes d'un fichier  
`perl -pe 's/^/$. /' fichier`

Supprime les 5 premiers caractères de chaque ligne  
`perl -pe '$_=substr($_,5)' fichier > fichier2`

Met le fichier en majuscule  
`perl -pe 'tr/a-z/A-Z/' fichier > fichier2`



## Filtres courts en Perl : option -p

Remplace le mot *vert* par *bleu*  
`perl -pe 's/\bvert\b/bleu/g' fichier > fichier2`

Conversion dos2unix  
`perl -pe 's/\r\n/\n/g' fichier_dos.txt`

Ne garde que le premier mot de chaque ligne  
`perl -pe 's/^\W*(\w+).*/$1/' fichier > fichier2`

Garde les 10 premières lignes d'un fichier  
`perl -pe 'exit if $.>10' fichier > fichier2`





## Filtres courts en Perl : option -i

Option `-i` : modification des fichiers «en ligne» (édition sur place)

Avant :

```
perl -pe 'tr/a-z/A-Z/' data.txt > data.new
mv -f data.new data.txt
```

Avec `-i` :

```
perl -i -pe 'tr/a-z/A-Z/' data.txt
⇒ Le fichier original est directement modifié
```

Sur plusieurs fichiers à la fois :

```
perl -i -pe 'tr/a-z/A-Z/' data.txt data2.txt
perl -i -pe 'tr/a-z/A-Z/' *.txt
```



## Filtres courts en Perl : option -i

L'option `-i` peut prendre un argument

⇒ utilisé pour créer un fichier de sauvegarde

Mode 1 : extention

```
perl -i.bak -pe 'tr/a-z/A-Z/' data.txt
data.txt : fichier modifié
data.txt.bak : fichier original
perl -i.bak -pe 'tr/a-z/A-Z/' *.txt
```

Mode 2 : motif libre avec `*`

```
perl -i 'save_*' -pe 'tr/a-z/A-Z/' *.txt
perl -i 'save/*.bak' -pe 'tr/a-z/A-Z/' *.txt
```



## Filtres courts en Perl : option -M

Option `-M` : chargement d'un module

Exemple : `perl -MData::Dumper -e 'print Dumper(1);'`

Décoder un fichier encodé en base64

```
perl -MMIME::Base64 -pe '$_=decode_base64($_)' fichier.b64
```

De latin1 à utf8 (et inversement) :

```
perl -MUnicode::String=latin1 -pi.lat1 -e '$_=latin1($_)->utf8'
perl -MUnicode::String=utf8 -pi.utf8 -e '$_=utf8($_)->latin1'
```

Appel de modules maison :

```
perl -MMonModule -e 'display(parse("passwd"))'
```



## Filtres courts en Perl : d'autres options

`-l` (comme «L»ine) : `chomp` sur chaque ligne lue  
et ajoute `\n` à la fin de chaque ligne affichée

`-I` : ajoute des répertoires dans `@INC`

`-0` (zéro) : modifie la notion de ligne avec `$/`

`-a` : ajout d'un split `/\s+/` sur chaque ligne (valeurs dans `@F`)

`-F` : changer le motif du split de `-a`

etc

Plus d'informations :

`perldoc perlrun`

<http://articles.mongueurs.net/magazines/linuxmag50.html>



### Lancer une commande : system

- `system '/usr/bin/prog';`  
`system 'ls -al *.txt';`

La sortie standard est la même que le script Perl.  
Les erreurs s'affichent également.

Attention : `and` pour tester

```
system 'ls -al *.txt' and die "erreur: $!";
```

Attente de la fin de la commande avant d'exécuter la suite du programme



### Lancer une commande : antiquotes

- `$sortie = '/usr/bin/prog';`  
En contexte scalaire : récupération en une seule chaîne de caractères de la sortie standard de la commande  
`$v = `df -h`;`
- `@lignes = '/usr/bin/prog';`  
`foreach('/usr/bin/prog') {...}`  
En contexte de liste : récupération de chaque ligne de la sortie standard de la commande  

```
foreach my $ligne (`df -h`) {
 print $ligne;
}
```
- Attente de la fin de la commande avant d'exécuter la suite du programme



### Lancer une commande : open

- `open(filedescr, 'commande')` exécution de la commande  
On lira sa sortie standard dans le descripteur de fichier.

```
open(my $fd, 'ls|') or die "$!";
open(my $fd, "df -HT $device|") or die "$!";
```

```
while(defined(my $ligne=<$fd>)) {
 print $ligne;
}
```

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



### Lancer une commande : open

- `open(filedescr, '|commande')` exécution de la commande  
On écrira son entrée standard dans le descripteur de fichier.  
Attention au signal SIGPIPE reçu si la commande se termine avant que nous ayons fini de lui fournir à lire

```
open(my $fd, "|gzip > $x.gz") or die "$!";
open(my $fd, '|mail robert@bidochon.org') or die "$!";
```

```
print $fd "chaîne\n";
```

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



## Lancer une commande : contrôle

- `system 'commande' and die "$!";`  
Attention : `and`
- `open(my $fd, 'commande|') or die "$!";`
- La variable `$?` contient des informations sur le dernier processus  
`$status = $? >> 8;` valeur de sortie  
`$signal = $? & 127;` signal qui a tué le processus  
`$coredp = $? & 128;` vrai si suite à un core dump

NB: La variable `$PATH` du shell est utilisée. NB en Perl : `$ENV{PATH}`



## Lancer une commande : aller plus loin

En Perl, on a aussi accès aux appels système `fork` et `exec`.

Un gestionnaire de fork : `Parallel::ForkManager`

Lancer des processus à partir de Perl

<http://articles.mongueurs.net/magazines/linuxmag55.html>



## Fonction eval

Cette fonction évalue les chaînes de caractères comme du code Perl.

- `eval chaîne` (coûteux)  
`my ($x,$y,$r) = (4,6);`  
`my $s = '$r = $x + $y';`  
`eval $s;`  $\Rightarrow$  `$r` vaut 10
- `$@` : erreur du dernier `eval`  
`eval '$r = $x / $y';`  
`warn($@) if($@);`

À utiliser avec parcimonie.



## Exemple : le rename de Larry Wall (allégé)

Usage : `rename 's/ancien(\..*)/nouveau$1/' *`  
`rename 'tr/A-Z/a-z/' *.txt`

```
#!/usr/bin/perl
$op = shift @ARGV;
foreach(@ARGV) {
 $was = $_;
 eval $op;
 die($@) if($@);
 rename($was,$_)
 if($was ne $_);
}
```

# `$_` vaut chaque nom de fichier  
# Sauvegarde du nom  
# Modification de `$_`  
# Gestion d'erreur  
# Renommage