perlopentut

Table des matières

1 NAME/NOM

perlopentut - Apprenez à ouvrir (des fichiers) à la mode Perl

2 DESCRIPTION

En Perl il existe deux manières standard d'ouvrir des fichiers : la syntaxe à la shell intéressante pour son coté pratique et la syntaxe C pour sa précision. La syntaxe shell se décline elle-même en formes à 2- et 3-paramètres dont la sémantique diffère. Vous avez le choix.

3 Open à la shell

La fonction `open` de Perl a été conçue pour imiter le fonctionnement des opérateurs de redirection du shell. Voici quelques exemples simples en shell :

```
$ monprogramme fichier1 fichier2 fichier3
$ monprogramme    <  fichiersource
$ monprogramme    >  fichiercible
$ monprogramme    >> fichiercible
$ monprogramme    |  autreprogramme
$ autreprogramme  |  monprogramme
```

Et voici quelques exemples plus complexes :

```
$ autreprogramme      | monprogramme f1 - f2
$ autreprogramme 2>&1 | monprogramme -
$ monprogramme     <&3
$ monprogramme     >&4
```

Les programmeurs habitués aux formes précédentes apprécieront d'apprendre que Perl supporte directement ces constructions famillières en utilisant une syntaxe quasiment identique à celle du shell.

3.1 Opens simples

La fonction `open` prend deux arguments : le premier est un manipulateur de fichier (appelé aussi handle de fichier ou filehandle) et le second est une simple chaîne de caractères indiquant à la fois quel fichier ouvrir et comment l'ouvrir. `open` retourne vrai quand tout s'est bien passé, et faux en cas d'échec en positionnant la variable interne `$!` pour indiquer quelle erreur système a provoqué l'échec. Si le handle du fichier était ouvert avant l'appel, il est implicitement fermé avant réouverture.

Quelques exemples :

```
open(INFO,"fichierdedonnees")
    || die("l'ouverture du fichierdedonnees a échoué: $!");
open(INFO,"<  fichierdedonnees")
    || die("l'ouverture du fichierdedonnees a échoué: $!");
open(RES, ">  fichierstats")
    || die("l'ouverture du fichierstats a échoué: $!");
open(LOG, ">> fichierlog ")
    || die("l'ouverture du fichierlog a échoué:  $!");
```

Si vous préférez économiser la ponctuation, vous pouvez aussi l'écrire ainsi :

```
open INFO,"<  fichierdedonnees"
    or die "l'ouverture du fichierdedonnees a échoué: $!";
open RES, ">  fichierstats"
    or die "l'ouverture du fichierstats a échoué: $!";
open LOG, ">> fichierlog "
    or die "l'ouverture du fichierlog a échoué: $!";
```

Il y a quelques détails à remarquer. D'abord, le symbole inférieur initial est optionnel. S'il est omis, Perl suppose que vous voulez ouvrir le fichier en lecture.

Remarquez aussi que le premier exemple utilise l'opérateur logique ||, tandis que le second utilise `or`, dont la précédence est plus faible. L'utilisation de || dans la seconde série d'exemples signifie en fait :

```
open INFO, ( "<  fichierdedonnees" || die "l'ouverture a échoué: $!" );
```

qui n'est certainement pas ce que vous souhaitez.

L'autre point important à noter est que, comme sous un shell Unix, les espaces qui suivent ou précèdent le nom de fichier sont ignorés. C'est une bonne chose car vous ne voudriez pas que les commandes qui suivent aient des comportements différents :

```
open INFO,   "<fichier"
open INFO,   "< fichier"
open INFO,   "<  fichier"
```

Ignorer les espaces entourant le nom aide également quand vous lisez ce nom depuis un autre fichier et oubliez de nettoyer les blancs indésirables avant l'ouverture :

```
$filename = <INFO>;         # houps, \n est encore là
open(EXTRA, "< $fichier") || die "l'ouverture de $filename a échoué: $!";
```

Ce n'est pas un bug, c'est volontaire. `open` imite le shell dans son utilisation des catactères de redirection pour spécifier le type d'ouverture désiré, il se comporte également comme le shell dans le traitement des espaces surnuméraires. Pour accéder à des fichiers portant des noms ennuyeux, allez-lire Exorciser la magie (§5.2).

Il existe aussi une version à trois paramètres de `open`, qui vous laisse isoler le caractère de redirection dans son propre champ :

```
open( INFO, ">", $fichier ) || die "Impossible de créer $fichier: $!";
```

Dans ce cas le nom du fichier à ouvrir est précisément la chaîne de caractères contenue dans `$fichier`, vous n'avez donc plus à vous soucier de savoir si `$fichier` contient des caractères susceptibles d'influencer le mode d'ouverture, ou des espaces au début du nom de fichier qui seraient absorbés dans un open à deux paramètres. Diminuer les interpolations inutiles de chaînes est aussi une bonne chose.

3.2 Handles de fichier indirects

Le premier paramètre de `open` peut être une référence vers un handle de fichier. Depuis perl 5.6.0, si ce paramètre n'est pas initialisé, Perl crée automatiquement un handle de fichier et place une référence vers ce handle dans le premier paramètre, comme ça :

```
open( my $in, $infile )  or die "Pas moyen d'ouvrir $infile: $!";
while ( <$in> ) {
    # utiliser $_
}
close $in;
```

Les handles de fichiers indirects facilitent la gestion des espaces de noms. Puisque les handles de fichiers sont globaux au paquetage courant, deux sous-routines qui essayeraient d'ouvrir `INFILE` entreraient en conflit. Avec deux fonctions ouvrant des handles de fichier indirects du genre `my $infile`, il n'y a plus de risque et il est inutile de s'inquiéter de futurs conflits hypothétiques.

Un autre comportement bien pratique est qu'un handle de fichier indirect est automatiquement fermé quand il sort de la portée courante ou quand on affecte undef à la variable :

```
sub premiereligne {
    open( my $in, shift ) && return scalar <$in>;
    # pas de close() nécessaire
}
```

3.3 L'ouverture des tubes

En C, quand vous voulez ouvrir un fichier en utilisant la librairie d'E/S standard vous utilisez la fonction `fopen`. Et pour ouvrir un tube vous utilisez la fonction `popen`. Par contre en shell vous utilisez simplement un caractère de redirection différent. C'est aussi la méthode retenue en Perl. L'appel à `open` reste le même ; c'est juste son paramètre qui diffère.

Si le premier caractère du second paramètre d'open est un symbole tube (une barre verticale), `open` lance une nouvelle commande et ouvre un handle de fichier en écriture seule qui transmet des données à cette commande. Cela vous permet d'écrire dans ce handle afin d'envoyer ce que vous voulez à l'entrée standard de la commande. Exemple :

```
open(IMPRIMANTE, "| lpr -Plp1")    || die "Erreur d'exécution de lpr: $!";
print IMPRIMANTE "machin\n";
close(IMPRIMANTE)                  || die "Erreur de fermeture de lpr: $!";
```

Si le dernier caractère du second paramètre d'open est un tube, vous lancez une nouvelle commande en ouvrant un handle de fichier en lecture seule associé à la sortie de cette commande. Via ce handle de fichier, vous pouvez ainsi lire tout ce que la commande produit sur sa sortie standard. Exemple :

```
open(RESEAU, "netstat -i -n |") || die "Erreur d'exécution de netstat: $!";
while (<RESEAU>) { }            # utiliser les données reçues
close(RESEAU)                   || die "Erreur de fermeture de netstat: $!";
```

Que se passe-t-il si vous essayez d'ouvrir un tube depuis ou à partir d'une commande inexistante ? Dans la mesure du possible, Perl essaye de détecter l'échec et de positionner `$!` comme d'habitude. Toutefois, si la commande contient certains caractères spéciaux tels que > or * (les 'metacaractères'), Perl n'exécute pas directement la commande. À la place, Perl lance un shell, qui lui-même essaye d'exécuter la commande. Cela signifie que c'est le shell qui reçoit le code d'erreur. Dans un tel cas, l'appel à `open` indiquera uniquement un échec si Perl ne peut pas exécuter un shell. Voyez Comment capturer STDERR à partir d'une commande externe ? in *perlfaq8* pour voir comment traiter ce cas. Vous trouverez une seconde explication dans *perlipc*.

Si vous voulez ouvrir un tube bidirectionel, la bibliothèque IPC::Open2 s'en chargera pour vous. Consultez Communications bidirectionnelles avec un autre processus in *perlipc*.

3.4 Le fichier moins ou tiret

Suivant, encore une fois, l'exemple des utilitaires standard du shell Unix, la fonction `open` traite de manière particulière un fichier dont le nom est un tiret (le signe moins "-"). Ouvrir tiret en lecture signifie en réalité accèder à l'entrée standard. Ouvrir tiret en écriture signifie accèder à la sortie standard.

Si le tiret peut être utilisé en tant qu'entrée ou sortie par défaut, que va-t-il se passer si on ouvre un tube vers ou à partir d'un tiret ? Quelle commande par défaut sera lancée ? Réponse : le script même que vous êtes en train d'exécuter ! Il s'agit en réalité d'un `fork` dissimulé à l'intérieur d'un appel à `open`. Référez-vous à Ouvrir des tubes en toute sécurité in *perlipc* pour les détails.

3.5 Mélanger la lecture et l'écriture

Il est possible d'ouvrir un fichier à la fois en lecture et en écriture. Il suffit d'ajouter un "+" devant le symbole de redirection. Comme en shell l'utilisation du symbole "inférieur" ne crée jamais de nouveau fichier et ne peut ouvrir qu'un fichier pré-existant. À l'inverse utiliser un symbole "supérieur" détruit (tronque à longueur zéro) un éventuel fichier pré-existant ou crée un nouveau fichier s'il n'en existe pas encore de ce nom. L'ajout d'un "+" pour la lecture/écriture n'affecte pas le fonctionnement de la fonction il tronque toujours l'existant ou crée un nouveau fichier.

```
open(WTMP, "+< /usr/adm/wtmp")
    || die "Impossible d'ouvrir /usr/adm/wtmp: $!";

open(SCREEN, "+> lkscreen")
    || die "Impossible d'ouvrir lkscreen: $!";

open(LOGFILE, "+>> /var/log/applog")
    || die "Impossible d'ouvrir /var/log/applog: $!";
```

Le premier exemple ne crée jamais de nouveau fichier et le second détruit toujours un fichier existant. Le troisième exemple crée un nouveau fichier si nécessaire mais sans effacer un fichier existant, et permet de lire à n'importe quel point du fichier, mais toutes les écritures auront lieu en fin de fichier. En bref, le premier cas est assez banal, bien plus que le second et le troisième, qui sont presque toujours des erreurs. (Si vous connaissez le C, le signe plus dans le `open` de Perl dérive historiquement de celui du fopen(3S) du C, fonction qui est appelée au bout du compte).

En pratique, si vous avez besoin de mettre à jour un fichier, à moins de travailler sur un fichier binaire comme dans le cas de WTMP ci-dessus, vous aurez tout intérêt à utiliser une autre approche. Le bon choix c'est l'option ligne de commande **-i** de Perl. La commande suivante prend un ensemble de sources C, C++, yacc et fichiers d'en-tête et remplace tous les foo par bar dans le corps de ces fichiers, tout en laissant intacts les fichiers d'origine avec un ".orig" à la fin de leur noms :

```
$ perl -i.orig -pe 's/\bfoo\b/bar/g' *.[Cchy]
```

C'est la manière courte de pratiquer un petit manège de renommage de fichiers fréquemment employé par les développeurs qui est en pratique la meilleure façon de mettre à jour des fichiers texte. Elle minimise notamment les risques de pertes de données et facilite et accélère le traitement. Voyez la question numéro 2 dans *perlfaq5* pour plus de details.

3.6 Les filtres

L'une des utilisations les plus fréquentes de `open` est si discrète que vous ne l'avez probablement jamais remarquée. Quand vous traitez les handle de fichier de ARGV en utilisant <ARGV>, Perl effectue en réalité un open implicite de chaque fichier de @ARGV. Ainsi un programme appelé par :

```
$ monprogramme fichier1 fichier2 fichier3
```

parviendra à ouvrir et traiter un par un tous les fichiers passés en ligne de commande en utilisant une construction très simple du genre :

```
while (<>) {
    # utiliser $_
}
```

Si @ARGV est vide au premier tour de boucle, Perl fait comme si vous aviez ouvert le fichier moins, c'est-à-dire l'entrée standard. En fait, $ARGV, le fichier en cours d'utilisation lors du traitement de <ARGV>, est même fixé à "-" dans de telles circonstances.

Vous êtes invité à pré-traiter le contenu d'@ARGV avant d'entamer la boucle pour vous assurer que son contenu vous convient. Une raison de procéder ainsi peut être par exemple de supprimer d'@ARGV les options de ligne de commande débutant par un signe moins. Même si rien ne vous empêche de gérer les cas simple à la main, simplifiez vous la vie avec la famille de modules getopts :

```
use Getopt::Std;

# -v, -D, -o ARG, positionnent $opt_v, $opt_D, $opt_o
getopts("vDo:");

# -v, -D, -o ARG, positionnent $args{v}, $args{D}, $args{o}
getopts("vDo:", \%args);
```

Ou le module standard Getopt::Long qui autorise les noms de paramètres longs :

```
use Getopt::Long;
GetOptions( "verbose"  => \$verbose,        # --verbose
            "Debug"    => \$debug,          # --Debug
            "output=s" => \$output );
       # --output=quelquechose ou --output quelquechose
```

Une autre raison de préparer la liste d'arguments peut être de choisir comme convention qu'une liste d'arguments vide signifie "tous les fichiers du répertoire courant";

```
@ARGV = glob("*") unless @ARGV;
```

Vous pouvez même choisir de conserver exclusivement les paramètres correspondant à des fichiers texte normaux. La technique ci-dessous est un peu laconique et vous pouvez bien entendu choisir d'afficher au passage les noms des fichiers retenus :

```
@ARGV = grep { -f && -T } @ARGV;
```

Si vous utilisez les options en ligne de commande **-n** ou **-p** de Perl, les modifications à @ARGV devront être placées dans un bloc `BEGIN{}`.

Souvenez-vous qu'un `open` normal n'a aucune propriété particulière, dans la mesure où il pourrait aussi bien appeller fopen(3S) ou popen(3S), en fonction de ses arguments ; c'est la raison pour laquelle on l'appelle parfois le "open magique". Voici un exemple :

```
$pwdinfo = `domainname` =~ /^(\(none\))?$/
                ? '< /etc/passwd'
                : 'ypcat passwd |';

open(PWD, $pwdinfo)
            or die "Impossible d'ouvrir $pwdinfo: $!";
```

Ce type de construction est aussi utilisée pour l'écriture de filtres. Dans la mesure où le traitement de <ARGV> utilise le `open` normal de Perl (avec la syntaxe à la mode shell) il profite de toutes les particularités que nous avons pu voir. Exemple :

```
$ monprogramme f1 "cmd1|" - f2 "cmd2|" f3 < tmpfile
```

Ce programme lit des données depuis le fichier *f1*, the processus *cmd1*, l'entrée standard (*tmpfile* ici puisqu'elle est redirigée), le fichier *f2*, la commande *cmd2*, et enfin fichier *f3*.

Oui, cela signifie aussi que si vous avez des fichiers nommés "-" (ou du même accabit) dans votre répertoire, `open` ne les traitera pas comme des littéraux. Vous allez devoir passer des paramètres du genre "./-", comme vous le feriez pour le programme *rm*, vous pouvez aussi choisir d'utiliser `sysopen` de la manière décrite ci dessous.

Une application très intéressante consiste à remplacer les fichiers portant certains noms par des tubes. Par exemple pour la décompression automatique via *gzip* des fichiers gzippés ou compressés.:

```
@ARGV = map { /^\.(gz|Z)$/ ? "gzip -dc $_ |" : $_  } @ARGV;
```

Ou encore, si vous avez installé le programme *GET* qui vient avec LWP, vous pouvez télécharger les URLs avant de les traiter.

```
@ARGV = map { m#^\w+://# ? "GET $_ |" : $_ } @ARGV;
```

Ce n'est pas pour rien qu'on appelle ça la magie de <ARGV>. Drôlement chiadé, non ?

4 Ouverture à la C

Si vous désirez la convivialité du shell alors choisissez sans hésiter, l'`open` façon Perl. En revanche si vous avez besoin de la finesse de contrôle qu'offre le simplissime fopen(3S) du C vous devriez jeter un coup d'oeil à la fonction Perl `sysopen`, une simple interface vers l'appel système open(2). Cela signifie qu'il est légérement plus tarabiscoté, mais c'est le prix de la précision.

`sysopen` prend 3 (ou 4) arguments.

```
sysopen HANDLE, PATH, FLAGS, [MASK]
```

Le paramètre HANDLE est un handle de fichier, exactement comme pour `open`. Le PATH est littérallement un chemin d'accès, qui ne tient donc aucun compte des caractères supérieur, inférieur, tube ou moins qu'il peut contenir et qui n'ignore pas non plus les espaces. Si un tel caractère est là c'est qu'il fait partie du chemin d'accès. Le paramètre FLAGS contient une ou plusieurs valeurs issues du module Fcntl et combinés par l'opérateur ou bit-à-bit "|".

Le dernier paramètre, le MASK, est optionnel ; s'il est présent, il est combiné avec l'umask courant de l'utilisateur pour générer le mode de création du fichier. Vous pouvez généralement l'omettre.

Bien que traditionnellemnt la lecture seule, l'écriture seule ou la lecture écriture soient respectivement associées aux valeurs 0, 1, et 2, ce n'est pas le cas sur certains systèmes d'exploitation. Il est donc préférable de charger le module Fcntl, qui fournit les constantes symboliques standard suivantes pour le paramètre FLAGS :

```
O_RDONLY            Lecture seule
O_WRONLY            Écriture seule
O_RDWR              Lecture et Écriture
O_CREAT             Créer le fichier s'il n'existe pas
O_EXCL              Échouer si le fichier existe
O_APPEND            Ajout à la fin du fichier
O_TRUNC             Tronquer le fichier
O_NONBLOCK          Accès non bloquant
```

Les constantes moins courantes suivantes sont disponibles ou non selon le système d'exploitation `O_BINARY`, `O_TEXT`, `O_SHLOCK`, `O_EXLOCK`, `O_DEFER`, `O_SYNC`, `O_ASYNC`, `O_DSYNC`, `O_RSYNC`, `O_NOCTTY`, `O_NDELAY` and `O_LARGEFILE`. Consultez la page de manuel open(2) ou son équivalent local pour les détails. (Note : à partir de Perl version 5.6 la constante `O_LARGEFILE`, est automatiquement ajoutée à sysopen() si elle est disponible. En effet, les grands fichiers sont le défaut.)

Voici comment utiliser `sysopen` pour émuler certains des appels à `open` que nous avons montré. Nous omettrons la gestion d'erreur `|| die $!` pour ne pas compliquer les exemples, mais assurez vous de bien tester les valeurs de retour dans du code de production. Les exemples présentés ne sont pas exactement identiques aux précédents dans la mesure ou `open` supprime les espaces en queue et en tête ce que ne fait pas `sysopen`, mais cela vous donnera l'idée générale.

Pour ouvrir un fichier en lecture :

```
open(FH, "< $path");
sysopen(FH, $path, O_RDONLY);
```

Pour ouvrir un fichier en écriture, créer un nouveau fichier si nécessaire ou tronquer un vieux fichier :

```
open(FH, "> $path");
sysopen(FH, $path, O_WRONLY | O_TRUNC | O_CREAT);
```

Pour ouvrir un fichier en ajout, en le créant si nécessaire :

```
open(FH, ">> $path");
sysopen(FH, $path, O_WRONLY | O_APPEND | O_CREAT);
```

Pour ouvrir un fichier en mise à jour, le fichier devant déjà exister :

```
open(FH, "+< $path");
sysopen(FH, $path, O_RDWR);
```

Et voici ce que l'on peut faire avec `sysopen` qui est impossible avec le `open` standard. Comme vous pouvez le constater il ne s'agit que de contrôler l'indicateur figurant en troisième argument :

Pour ouvrir un fichier en écriture en créant un nouveau fichier, celui-ci ne devant pas préexister :

```
sysopen(FH, $path, O_WRONLY | O_EXCL | O_CREAT);
```

Pour ouvrir un fichier en ajout, ce fichier devant déjà exister :

```
sysopen(FH, $path, O_WRONLY | O_APPEND);
```

Pour ouvrir un fichier en mise à jour, en créant un nouveau fichier si nécessaire :

```
sysopen(FH, $path, O_RDWR | O_CREAT);
```

Pour ouvrir un fichier en mise à jour, ce fichier ne devant pas préexister :

```
sysopen(FH, $path, O_RDWR | O_EXCL | O_CREAT);
```

Pour ouvrir un fichier en mode non bloquant, en le créant si nécessaire :

```
sysopen(FH, $path, O_WRONLY | O_NONBLOCK | O_CREAT);
```

4.1 Permissions à la mode Unix

Si vous négligez le paramètre MASK de `sysopen`, Perl utilise la valeur octale 0666. Le zéro initial indique qu'il s'agit d'un nombre écrit en octal, le premier chiffre octal correspond aux droits du propriétaire du fichier, le second aux droits de son groupe et le troisième à ceux des autres. Le MASK normal pour les exécutables et les répertoires devrait être 0777, et pour tout le reste 0666.

Pourquoi une telle permissivité ? En fait cette façon de faire est loin d'être aussi permissive qu'il peut sembler. En effet comme nous l'avons dit le MASK est modifié par l'`umask` du processus actif. L'umask est un nombre représentant les bits de permission *désactivés*; c'est-à-dire les bits qui ne doivent pas être activés même s'ils sont explicitement demandés dans les permissions des fichiers créés.

Par exemple, si votre `umask` vaut 027, alors la partie 020 désactive les droits d'écriture pour le groupe, et la partie 007 empêche les autres de lire, écrire ou exécuter des fichiers. Dans un tel cas passer 0666 à `sysopen` crée un fichier avec le mode 0640, puisque `0666 & ~027` vaut 0640.

Vous n'aurez que très rarement besoin d'utiliser le paramètre MASK de `sysopen()`. En l'utilisant vous privez l'utilisateur de la liberté de choisir les droits des nouveaux fichiers. Forcer un choix est presque toujours mauvais. Une exception pourrait être le stockage de données sensibles ou privées, comme le mail, les fichiers de cookies et les fichiers temporaires internes.

5 Trucs obscurs avec open

5.1 Ré-ouverture de fichiers (dups)

De temps en temps on dispose déjà d'un handle vers un fichier ouvert et on veux obtenir un second handle identique au premier. Pour faire ce genre de choses en shell on place un et-commercial "&" devant le numéro du descripteur au niveau des redirections. Par exemple, `2>&1` prend le descripteur 2 (qui correspond à STDERR en Perl) et le redirige vers le descripteur 1 (qui correspond en général à STDOUT). Le principe est fondamentalement le même en Perl : un nom de fichier commençant par un et-commercial est traité comme un numéro de descripteur de fichier s'il s'agit d'un nombre ou comme un handle de fichier s'il s'agit d'une chaine.

```
open(SAUVEOUT, ">&SAUVEERR") || die "Impossible de dupliquer SAUVEERR: $!";
open(MHCONTEXTE, "<&4")      || die "Impossible de dupliquer fd4: $!";
```

Cela signifie que si une fonction attend un nom de fichier mais que vous ne voulez ou ne pouvez pas lui indiquer un nom de fichier parce que le fichier en question est déjà ouvert, vous pouvez vous contenter du handle de fichier précédé d'un &-commercial. Il est tout de même préférable d'utiliser un handle pleinement qualifié, juste au cas ou la fonction serait dans un autre paquetage :

```
unefunction("&main::FICHIERLOG");
```

De cette manière si unefunction() à l'intention d'ouvrir son paramètre il peut se contenter d'utiliser le handle déjà ouvert. Ce n'est pas la même chose que de passer un handle. En effet, dans le cas d'un handle vous n'avez pas la possibilité d'ouvrir le fichier. Ici vous avez l'opportunité de transmettre quelque chose à open.

Si vous disposez de l'un de ces complexes objets d'E/S C++, très à la mode dans certains milieux, alors cette technique ne fonctionne pas car ces objets ne sont pas des handle de fichier corrects dans le sens Perl. Vous allez devoir faire un appel à fileno() pour en extraire le véritable numéro de descripteur, si vous pouvez :

```
use IO::Socket;
$handle = IO::Socket::INET->new("www.perl.com:80");
$fd = $handle->fileno;
unefunction("&$fd");  # ce n'est pas un appel de fonction indirect
```

Il peut cependant être plus facile (et cela sera certainement plus rapide) d'utiliser simplement de vrais handles de fichiers :

```
use IO::Socket;
local *REMOTE = IO::Socket::INET->new("www.perl.com:80");
die "Impossible de se connecter" unless defined(fileno(REMOTE));
unefunction("&main::REMOTE");
```

Si le handle de fichier ou le numéro de descripteur est précédé non pas d'un simple "&" mais plutôt d'une combinaison "&=", alors Perl ne crée pas tout à fait un nouveau descripteur ouvert sur le même fichier en utilisant l'appel système dup(2). Ce qui est créé est plutôt une sorte d'alias vers le descripteur existant en utilisant la bibliothèque système fdopen(3S). Cette technique est légèrement plus économique en ressources systèmes, même si c'est de moins en moins une préoccupation à notre époque. Voici ce que cela donne :

```
$fd = $ENV{"MHCONTEXTFD"};
open(MHCONTEXT, "<&=$fd")   or die "Échec de l'appel à fdopen $fd: $!";
```

Si vous utilisez le <ARGV> magique, vous pourriez même utiliser cette syntaxe en tant qu'argument de ligne de commande de @ARGV, quelque chose du genre "<&=$MHCONTEXTFD", mais personne ne l'a encore fait à notre connaissance.

5.2 Exorciser la magie

Perl est un langage plus intuitif que, disons Java. Intuitif signifie que les choses se passent généralement comme on le souhaite, même si l'on en comprend pas toujours le pourquoi du comment et que cela ressemble dans certains cas au lapin sortant du chapeau. Malheureusement cette approche conduit de temps à autres à ce qu'il se passe un peu trop de choses en coulisses. La magie de Perl tourne alors à la malédiction.

Si la version magique de `open` est un peu trop magique à votre gout, vous n'êtes pas obligé de vous résigner à utiliser `sysopen`. Pour ouvrir un fichier contenant des caractère spéciaux, il faut protéger les espaces en tête et en fin de nom. Les espaces en tête sont protégés en ajoutant simplement "./" en tête du nom de fichier débutant par des espaces. Les espaces en fin de nom sont protégés en ajoutant explicitement un octet ASCII NUL ("\0") en fin de chaine.

```
$fichier =~ s#^(\s)#./$1#;
open(FH, "< $fichier\0")   || die "Impossible d'ouvrir $fichier: $!";
```

Cette technique suppose, bien sûr, que sur votre système, le point désigne le dossier courant, le slash le séparateur de dossier et que les octets ASCII NUL sont interdits dans les noms de fichiers valides. La majorité des systèmes d'exploitation acceptent ces conventions, aussi bien les systèmes POSIX que les systèmes propriétaires Microsoft. L'unique système relativement répandu qui ne fonctionne pas ainsi est le système Macintosh "Classique", qui utilise les deux points (:) là où les autres ont opté pour le slash. Peut-être que `sysopen` n'est peut-être pas une si mauvaise idée après tout.

Si vous voulez traiter <ARGV> de façon mortellement ennuyeuse et non magique, vous pouvez commencer par faire ceci :

```
#   "Sam s'assit par terre et pris sa tête dans ses mains.
#   'Je souhaite ne jamais être venu ici et je ne veux plus jamais
#   voir de magie' dit-il, avant de se taire."
for (@ARGV) {
    s#^([^./])#./$1#;
    $_ .= "\0";
}
while (<>) {
    # now process $_
}
```

Soyez conscient que les utilisateurs ne vont pas apprécier d'être privé du "-" pour désigner l'entrée standard comme ils en ont l'habitude.

5.3 Chemins d'accès comme opens

Vous avez sans doute remarqué le format de certains messages produits par les fonctions `warn` et `die` de Perl :

```
Un avertissement at nom_du_script line 29, <FH> line 7.
```

Le message signifie que vous avez ouvert un handle de fichier FH et lus sept enregistrements à partir de FH. Mais quel était donc le nom du fichier associé à ce handle ?

Si vous n'avez pas activé l'option `strict refs`, ou si vous l'avez temporairement désactivée, alors il vous suffit d'écrire :

```
open($chemin, "< $chemin") || die "Impossible d'ouvrir $chemin: $!";
while (<$chemin>) {
    # du code
}
```

Puisque vous utilisez le chemin d'accès au fichier en tant que handle, vos avertissements ressembleront à :

```
Un avertissement at nom du script line 29, </etc/motd> line 7.
```

5.4 Open à un seul argument

Vous vous souvenez que nous avons dit que le open de Perl prenait deux arguments. En réalité nous avons fait preuve d'une certaine mauvaise foi. C'est un mensonge par omission. En réalité open peut aussi ne prendre qu'un seul argument. À la condition exclusive d'utiliser une variable globale pour le handle de fichier, avec un lexical ça ne marche pas, vous pouvez ne passer à `open` qu'un seul paramètre, le handle de fichier, le nom de fichier sera extrait de la variable scalaire globale de même nom.

```
$FICHIER = "/etc/motd";
open FICHIER or die "Impossible d'ouvrir $FILE: $!";
while (<FICHIER>) {
    # du code
}
```

La raison de cette étrangeté ? Perl aussi comporte son lot d'horreurs hyst[éo]riques. Ce comportement existe dans Perl quasiment depuis sa création, voire même avant.

5.5 Jouer avec STDIN et STDOUT

Une bonne idée avec STDOUT c'est de le fermer explicitement lorsque votre programme n'en a plus besoin.

```
END { close(STDOUT) || die "Impossible de fermer stdout: $!" }
```

Si vous ne le faites pas et que votre programme remplit la partition courante à cause d'une redirection de ligne de commande, l'erreur n'apparaitra pas et le programme ne renverra pas de statut d'échec en sortie.

Vous n'êtes pas obligé d'accepter STDIN et le STDOUT tels qu'on vous les donne. Rien ne vous empêche de les réouvrir si vous le souhaitez :

```
open(STDIN, "< fichier")
    || die "Impossible d'ouvrir fichier: $!";

open(STDOUT, "> sortie")
    || die "Impossible d'ouvrir sortie: $!";
```

Et ces nouvelles E/S standard peuvent être accédées directement ou passées à d'autres processus. Tout se passe comme si le programme était appelé au départ avec ces redirections indiquées en ligne de commande.

Il est probablement plus intéressant de connecter les E/S standard à des tubes. Par exemple :

```
$pager = $ENV{PAGER} || "(less || more)";
open(STDOUT, "| $pager")
    || die "Impossible de forker un pager: $!";
```

Tout se passe comme si votre programme était appelé avec son stdout déjà redirigé vers votre pager. Vous pouvez aussi utiliser ce genre de choses en conjonction avec un fork implicite vers le script en cours. Vous pouvez avoir envie de faire ce genre de choses pour que votre programme gère lui-même le post traitement des données, mais dans un second processus :

```
head(100);
while (<>) {
    print;
}

sub head {
    my $lines = shift || 20;
    return if $pid = open(STDOUT, "|-");  # return si process père
    die "Échec du fork: $!" unless defined $pid;
    while (<STDIN>) {
        last if --$lines < 0;
        print;
    }
    exit;
}
```

Cette technique peut être itérée pour mettre en place autant de filtres successifs que vous le désirez sur le flux de sortie.

6 Autres considérations sur les E/S

Les sujets qui suivent ne sont pas directement liées aux paramètres de `open` ou de `sysopen`, mais ils agissent sur le comportement de vos fichiers ouverts.

6.1 Ouvrir des fichiers qui n'en sont pas

Quand un fichier n'est-il pas un fichier ? Une réponse possible consiste à dire qu'il existe mais que ce n'est pas un simple fichier. Nous vérifions tout d'abord s'il ne s'agit pas d'un lien symbolique, juste au cas où.

```
if (-l $file || ! -f _) {
    print "$file n'est pas un vrai fichier\n";
}
```

Quels sont les autres types de fichiers que (sic) les fichiers eux-mêmes ? Les répertoires, les liens symboliques, les tubes nommés, les sockets du domaine Unix et les périphériques blocs ou caractères. Tous sont des fichiers, mais pas de *simples* fichiers. Attention à ne pas tout confondre : la question n'est pas de savoir si ce sont des fichiers textes ou binaires. Il existe des fichiers textes qui ne sont pas de simples fichiers, et, inversement, il existe des fichiers simples qui ne sont pas des fichiers textes. C'est la raison pour laquelle il existe deux opérateurs unaires distincts de tests de fichiers `-f` et `-T`.

Pour ouvrir un répertoire, vous devriez utiliser la fonction `opendir`, puis la traiter avec `readdir`, en reconstruisant prudemment le chemin d'accès aux fichiers si nécessaire.

```
opendir(DIR, $dirname) or die "Échec d'opendir sur $dirname: $!";
while (defined($file = readdir(DIR))) {
    # faire quelque chose avec "$dirname/$file"
}
closedir(DIR);
```

Si vous voulez parcourir des dossiers récursivement mieux vaut utiliser le module File::Find. L'exemple suivant affiche tous les fichiers récursivement et ajoute un slash à leur nom si le fichier est un dossier.

```
@ARGV = qw(.) unless @ARGV;
use File::Find;
find sub { print $File::Find::name, -d && '/', "\n" }, @ARGV;
```

L'exemple suivant détecte tous les liens symboliques erronés dans l'arborescence d'un dossier particulier.

```
find sub { print "$File::Find::name\n" if -l && !-e }, $dir;
```

Comme vous pouvez le constater avec les liens symboliques vous pouvez prétendre que le lien est ce vers quoi il pointe. Ou bien, si vous voulez savoir *vers quoi* il pointe, appeler `readlink`:

```
if (-l $file) {
    if (defined($pouf = readlink($fichier))) {
        print "$fichier pointe vers $pouf\n";
    } else {
        print "$fichier pointe dans le vide: $!\n";
    }
}
```

6.2 Ouvrir des tubes nommés

Les tubes nommés sont une toute autre affaire. On y accède comme s'il s'agissait de fichiers normaux, mais leur ouverture est généralement bloquante tant qu'il n'y a pas de candidats pour y écrire et pour y lire. Vous pouvez en apprendre plus sur leur fonctionnement en lisant Tubes nommés in *perlipc*. Les sockets du domaine Unix sont encore un autre genre de bestiole ; ils sont décrits dans Clients et Serveurs TCP du Domaine Unix in *perlipc*.

Quant à l'ouverture de fichiers de périphériques, elle est tantôt facile tantôt plus épineuse suivant le cas. Nous supposerons que si vous ouvrez un périphérique de type "blocs", vous savez ce que vous êtes en train de faire. Les périphérique "caractères" méritent un peu plus d'attention. C'est ce type de périphérique qui est typiquement utilisé pour les modems et certains types d'imprimantes. C'est le sujet abordé dans Comment lire et écrire sur des ports série ? in *perlfaq8*. Il suffit en général de faire attention en les ouvrant :

```
sysopen(TTYIN, "/dev/ttyS1", O_RDWR | O_NDELAY | O_NOCTTY)
            # (O_NOCTTY n'est plus nécessaire sur les systèmes POSIX)
    or die "Impossible d'ouvrir /dev/ttyS1: $!";
open(TTYOUT, "+>&TTYIN")
    or die "Impossible de dupliquer TTYIN: $!";

$ofh = select(TTYOUT); $| = 1; select($ofh);

print TTYOUT "+++at\015";
$answer = <TTYIN>;
```

Pour les decripteurs que vous n'avez pas ouvert via `sysopen`, comme les sockets, il reste possible de forcer le mode non-bloquant en utilisant `fcntl`:

```
use Fcntl;
my $anciens_indicateurs = fcntl($handle, F_GETFL, 0)
    or die "Impossible de récupérer les indicateurs: $!";
fcntl($handle, F_SETFL, $anciens_indicateurs | O_NONBLOCK)
    or die "Impossible de passer en mode non bloquant: $!";
```

Plutôt que de perdre votre temps en vous enlisant dans les spécificités d'une foule d'`ioctl`s tous différents, si vous envisagez de manipuler des terminaux, alias *ttys*, mieux vaut faire appel au programme stty(1) s'il existe sur votre système, ou à défaut utiliser l'interface portable POSIX. Pour comprendre de quoi il retourne, vous devrez lire la page de man de termios(3) qui décrit l'interface POSIX vers les périphériques tty, et enfin *POSIX*, qui décrit l'interface Perl permettant d'accèder aux fonctions POSIX. Il existe aussi quelques modules de plus haut niveau sur CPAN qui peuvent vous aider. Voyez Term::ReadKey et Term::ReadLine.

6.3 Ouvrir des Sockets

Que reste-t'il encore à ouvrir ? Pour ouvrir une connexion par sockets, vous n'utiliserez aucun des deux open de Perl. Voyez pour cela Sockets : Communication Client/Server in *perlipc*. Voici un exemple. Une fois que vous l'avez vous pouvez utilisant FH comme n'importe quel handle de fichier bidirectionnel.

```
use IO::Socket;
local *FH = IO::Socket::INET->new("www.perl.com:80");
```

Pour ouvrir une URL suivez scrupuleusement l'ordonnance du docteur, une bonne cuillérée du module de CPAN LWP matin et soir. Ici aucun handle de fichier mais il reste très facile de récupérer le contenu d'un document :

```
use LWP::Simple;
$doc = get('http://www.linpro.no/lwp/');
```

6.4 Fichiers binaires

La lourde hérédité de certains systèmes leur fait utiliser des modèles d'entrées/sorties que les âmes charitables qualifient de mal fichus et tortueux (d'autres disent simplement défectueux). Un fichier n'est pas un fichier, du moins pas du point de vue des librairies d'E/S standard du C. Sur ces vieux systèmes sur lesquels les bibliothèques C (mais pas le noyau) font la différence entre les flux textes et binaires, quelques acrobaties sont nécessaires pour ne pas avoir d'ennuis avec les fichiers. Sur ces malencontreux systèmes les sockets et les tubes sont ouverts d'emblée en mode binaire, et il n'existe à ce jour aucun moyen de faire autrement. Dans le cas des fichiers vous avez un peu plus de marge de manoeuvre :

Vous pouvez appliquer la fonction `binmode` aux handles concernés avant tout accès aux fichiers :

```
binmode(STDIN);
binmode(STDOUT);
while (<STDIN>) { print }
```

Une alternative, sur les systèmes qui le gèrent, consiste à transmettre à `sysopen` une option non standard pour ouvrir le fichier en mode binaire. La solution est équivalente à une ouverture normale du fichier suivie d'un appel à `binmode` sur le handle.

```
sysopen(BINDAT, "fichier.data", O_RDWR | O_BINARY)
    || die "Impossible d'ouvrir fichier.data: $!";
```

Une fois cela fait vous pouvez utiliser `read` et `print` sur ce handle sans crainte que la bibliothèque d'entrées/sorties non standard détruise vos données. Ce n'est pas folichon, mais les tares héréditaires le sont rarement. CP/M restera avec nous jusqu'à la fin des temps... et après.

Sur les systèmes utilisant des couches d'E/S exotiques, il s'avère que, étonnament, même les E/S non bufferisées qui utilisent `sysread` et `syswrite` peuvent discrètement mutiler vos données dès que vous avez le dos tourné.

```
while (sysread(WHENCE, $buf, 1024)) {
    syswrite(WHITHER, $buf, length($buf));
}
```

Suivant les vicissitudes de votre plate-forme d'exécution, même ces appels peuvent exiger un détour préalable par `binmode` ou `O_BINARY`. Il est de notoriété publique que Unix, MacOS, Plan 9 et et Inferno ne posent aucun problème.

6.5 Verrouillage de fichiers

Dans un environnement multitâches, il est parfois nécessaire d'être prudent afin d'éviter les conflits avec d'autres processus qui voudraient réaliser des E/S sur les fichiers auxquels vous êtes justement en train d'accèder.

Sur des fichiers en lecture et en écriture vous aurez souvent besoin de verrous, respectivement partagés ou exclusifs. Vous pouvez aussi faire comme s'il n'existait que des verrous exclusifs.

N'utilisez jamais l'opérateur d'existence d'un fichier `-e $fichier` en tant qu'indicateur de verrouillage, il existe en effet dans ce cas un problème d'accès concurrent (*race condition*) entre le test d'existence du fichier et sa création. Il est possible pour un autre processus de créer un fichier pendant la tranche de temps qui sépare votre test d'existence et votre tentative de création du fichier. Ici l'atomicité de l'opération est critique.

Le mécanisme de verrouillage la plus portable en Perl consiste à utiliser la simplissime fonction `flock`, qui est émulée sur les systèmes qui ne la supportent pas directement comme SysV et Windows. La sémantique sous-jacente peut affecter le fonctionnement d'ensemble, vous devriez donc chercher à savoir comment `flock` est implémentée sur le portage de Perl de votre système d'exploitation.

Le verrouillage d'un fichier *n'empêche pas* les autres processus d'accèder au fichier. Un verrou de fichier bloque uniquement les processus demandant à verrouiller ce fichier, et n'empèche en rien de réaliser des opérations d'E/S sur celui-ci. Donc dans la mesure où le système des verrous fonctionne de manière purement coopérative, si un processus les utilise mais que d'autres les ignorent, rien ne va plus.

Par défaut un appel à `flock` va bloquer jusqu'à ce qu'un verrou soit accordé. Une demande de verrou partagé est accordée dés lors qu'aucune demande de verrou exclusif n'est active. Un verrou exclusif n'est accordé que si aucun verrouillage d'aucune sorte n'est actif. Les verrous sont associés à des descripteurs de fichier et non à des noms de fichiers, vous ne pouvez donc pas verrouiller un fichier avant de l'avoir ouvert ni maintenir un verrou actif une fois le fichier fermé.

Voici comment obtenir un verrou bloquant partagé sur un fichier, typiquement pour des accès en lecture :

```
use 5.004;
use Fcntl qw(:DEFAULT :flock);
open(FH, "< fichier")  or die "Erreur d'ouverture de fichier: $!";
flock(FH, LOCK_SH)      or die "Erreur de verrouillage de fichier: $!";
# now read from FH
```

Vous pouvez aussi obtenir un verrou non bloquant en utilisant `LOCK_NB`.

```
flock(FH, LOCK_SH | LOCK_NB)
    or die "Erreur de verrouillage de fichier: $!";
```

L'exemple suivant montre comment obtenir un comportement plus convivial en avertissant l'utilisateur que vous allez bloquer :

```
use 5.004;
use Fcntl qw(:DEFAULT :flock);
open(FH, "< fichier")  or die "Erreur d'ouverture de fichier: $!";
unless (flock(FH, LOCK_SH | LOCK_NB)) {
    $| = 1;
    print "En attente du verrou...";
    flock(FH, LOCK_SH)  or die "Erreur de verrouillage du fichier: $!";
    print "Verrouillé.\n"
}
# maintenant lire sur FH
```

Pour obtenir un verrou exclusif, typiquement nécessaire pour l'écriture, un minimum de prudence s'impose. Nous ouvrons le fichier par `sysopen` pour qu'il soit verrouillé avant d'être vidé. Un équivalent non bloquant est possible en utilisant `LOCK_EX | LOCK_NB`.

```
use 5.004;
use Fcntl qw(:DEFAULT :flock);
sysopen(FH, "fichier", O_WRONLY | O_CREAT)
    or die "Erreur d'ouverture de fichier: $!";
flock(FH, LOCK_EX)
    or die "Erreur de verrouillage de fichier: $!";
truncate(FH, 0)
    or die "Erreur de troncature de fichier: $!";
# maintenant écrire dans le fichier
```

Finalement, pour satisfaire ces trop nombreux webmestres que nul n'est parvenu à dissuader de gâcher quantité de cycles machines pour faire fonctionner un petit gadget vaniteux baptisé "Compteur de visites", voici comment incrémenter un nombre à l'intérieur d'un fichier en toute sécurité :

```
use Fcntl qw(:DEFAULT :flock);

sysopen(FH, "fichiercompteur", O_RDWR | O_CREAT)
    or die "Erreur d'ouverture de fichiercompteur: $!";
# autoflush FH
$ofh = select(FH); $| = 1; select ($ofh);
flock(FH, LOCK_EX)
    or die "Erreur de verrouillage en écriture de fichiercompteur: $!";

$num = <FH> || 0;
seek(FH, 0, 0)
    or die "Erreur de positionnement au début de fichiercompteur: $!";
print FH $num+1, "\n"
    or die "Erreur d'écriture dans fichier compteur: $!";

truncate(FH, tell(FH))
    or die "Erreur de troncature de fichiercompteur: $!";
close(FH)
    or die "Erreur de fermeture de fichiercompteur: $!";
```

6.6 Couches d'entrées/sorties

Avec Perl 5.8.0 un nouveau paradigme d'entrées/sorties baptisé "PerlIO" a été introduit. Il s'agit d'une "tuyauterie" flambant neuve pour tous les événements d'E/S en Perl ; pour l'essentiel tout fonctionne exactement comme par le passé, mais PerlIO offre aussi quelques nouvelles possibilités, comme permettre de penser aux E/S en termes de filtres successifs.

Un filtre d'E/S ne se contente plus uniquement de déplacer les données mais peut également leur appliquer des transformations. Il peut s'agir de transformations de type compression/décompression, cryptage/décryptage, ou encore de transcodages entre différents jeux de caractères.

Une présentation complète de PerlIO sortirait du cadre de ce tutoriel, mais voici déjà quelques façons d'indiquer l'utilisation de filtres :

– En utilisant la forme à trois arguments de `open` on ajoute au second argument quelque chose de plus que les habituels `'<'`, `'>'`, `'>>'`, `'|'` et leurs variantes, par exemple :

```
open(my $fh, "<:crlf", $nomfichier);
```

– En utilisant la forme à deux arguments de `binmode` cela donne :

```
binmode($fh, ":encoding(utf16)");
```

Pour une discussion en profondeur de PerlIO voir *PerlIO*; Pour une discussion en profondeur d'Unicode et des E/S voir *perluniintro*.

7 VOIR AUSSI

Les fonctions `open` et `sysopen` dans *perlfunc*, les pages de manuel open(2), dup(2), fopen(3) et fdopen(3) ainsi que la documentation POSIX.

8 AUTEUR ET COPYRIGHT

9 TRADUCTION

9.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez http://perl.enstimac.fr/.

9.2 Traducteur

Christophe Grosjean <christophe.grosjean@gmail.com>

9.3 Relecture

Pas encore...

10 HISTORIQUE

Première version : Sat Jan 9 08:09:11 MST 1999

Traduction initiale : Mer Nov 8 11:41:00 GMT 2006

11 À propos de ce document

Ce document est la traduction française du document orginal distribué avec perl. Vous pouvez retrouver l'ensemble de la documentation française Perl (éventuellement mise à jour) en consultant l'URL <http://perl.enstimac.fr/>.

Ce document PDF a été produit Paul Gaborit. Si vous utilisez la version PDF de cette documentation (ou une version papier issue de la version PDF) pour tout autre usage qu'un usage personnel, je vous serai reconnaissant de m'en informer par un petit message <mailto:Paul.Gaborit@enstimac.fr>.

Si vous avez des remarques concernant ce document, en premier lieu, contactez la traducteur (vous devriez trouver son adresse électronique dans la rubrique TRADUCTION) et expliquez-lui gentiment vos remarques ou critiques. Il devrait

normalement vous répondre et prendre en compte votre avis. En l'absence de réponse, vous pouvez éventuellement me contacter.

Vous pouvez aussi participer à l'effort de traduction de la documentation Perl. Toutes les bonnes volontés sont les bienvenues. Vous devriez trouver tous les renseignements nécessaires en consultant l'URL ci-dessus.