

Table des Matières

1 NAME	1
2 DESCRIPTION	2
<u>2.1 Fonctions Perl par catégories</u>	3
<u>2.2 Portabilité</u>	5
<u>2.3 Fonctions Perl par ordre alphabétique</u>	5
3 TRADUCTION	94
<u>3.1 Version</u>	94
<u>3.2 Traducteur</u>	94
<u>3.3 Relecture</u>	94

1 NAME

perlfunc – Fonctions Perl prédéfinies

2 DESCRIPTION

Les fonctions de cette section peuvent être utilisées en tant que termes dans une expression. Elles se séparent en deux catégories principales : les opérateurs de listes et les opérateurs unaires nommés. Ceux-ci diffèrent dans leurs relations de priorité avec la virgule qui les suit. (Cf. la table de priorité dans [the perlop manpage](#).) Les opérateurs de liste prennent plusieurs arguments alors que les opérateurs unaires n'en prennent jamais plus d'un. Une virgule termine alors l'argument d'un opérateur unaire mais sépare les arguments d'un opérateur de liste. Un opérateur unaire fournit en général un contexte scalaire à son argument, alors qu'un opérateur de liste fournit un contexte, soit scalaire, soit de liste, pour ses arguments. S'il propose les deux, les arguments scalaires seront les premiers et la liste d'arguments suivra. (Notez qu'il ne peut y avoir qu'une seule liste d'arguments.) Par exemple, `splice()` a trois arguments scalaires suivis d'une liste alors que `gethostbyname()` a quatre arguments scalaires.

Dans la description syntaxique qui suit, les opérateurs de liste qui attendent une liste (et fournissent un contexte de liste pour les éléments de cette liste) ont pour argument LISTE. Une telle liste peut être constituée de toute combinaison de valeurs d'arguments scalaires ou de listes ; les valeurs de listes seront incluses dans la liste comme si chaque élément individuel était interpolé à cet emplacement de la liste, formant ainsi la valeur d'une longue liste unidimensionnelle. Les éléments de LISTE doivent être séparés par des virgules.

Toute fonction de la liste ci-dessous peut être utilisée avec ou sans parenthèses autour de ses arguments. (Les descriptions syntaxiques les omettent) Si vous utilisez les parenthèses, la simple (mais parfois surprenante) règle est la suivante : ça *RESSEMBLE* à une fonction, donc c'*EST* une fonction, et la priorité importe peu. Sinon, c'est un opérateur de liste ou un opérateur unaire et la priorité a son importance. Les espaces entre la fonction et les parenthèses ne comptent pas, vous devez donc faire parfois très attention :

```
print 1+2+4;          # affiche 7.
print(1+2) + 4;      # affiche 3.
print (1+2)+4;       # affiche aussi 3 !
print +(1+2)+4;      # affiche 7.
print ((1+2)+4);     # affiche 7.
```

Si vous exécutez Perl avec l'option `-w`, vous pourrez en être averti. Par exemple, la troisième ligne ci-dessus génère :

```
print (...) interpreted as function at - line 1.
Useless use of integer addition in void context at - line 1.
```

Quelques rares fonctions ne prennent aucun argument et ne sont donc ni des opérateurs unaires ni des opérateurs de liste. Cela inclut des fonctions telles que `time` et `endpwent`. Par exemple, `time+86_400` signifie toujours `time() + 86_400`.

Pour les fonctions qui peuvent être utilisées dans un contexte scalaire ou de liste, une erreur non fatale est généralement indiquée dans un contexte scalaire en retournant la valeur indéfinie, et dans un contexte de liste en retournant la liste nulle.

Rappelez-vous de l'importante règle suivante : il n'y a **aucune règle** qui lie le comportement d'une expression dans un contexte de liste à son comportement dans un contexte scalaire, et réciproquement. Cela peut générer deux résultats complètement différents. Chaque opérateur et

chaque fonction choisit le type de valeur qui semble le plus approprié de retourner dans un contexte scalaire. Certains opérateurs retournent la longueur de la liste qui aurait été retournée dans un contexte de liste. D'autres opérateurs retournent la première valeur. D'autres encore retournent la dernière valeur. D'autres enfin retournent le nombre d'opérations réussies. En général, ils font ce que vous souhaitez, à moins que vous ne vouliez de la consistance.

Un tableau nommé en contexte scalaire est assez différent de ce qui apparaîtrait au premier coup d'oeil comme une liste dans un contexte scalaire. Vous ne pouvez pas transformer une liste comme (1, 2, 3) dans un contexte scalaire, car le compilateur connaît le contexte à la compilation. Il générerait ici l'opérateur scalaire virgule, et non pas la version construction de liste de la virgule. Ce qui signifie que ça n'a jamais été considéré comme une liste avec laquelle travailler.

En général, les fonctions en Perl qui encapsulent les appels système du même nom (comme `chown(2)`, `fork(2)`, `closedir(2)`, etc.) retournent toutes vrai quand elles réussissent et `undef` sinon, comme c'est souvent mentionné ci-dessous. C'est différent des interfaces C qui retournent `-1` en cas d'erreur. Les exceptions à cette règle sont `wait`, `waitpid()` et `syscall()`. Les appels système positionnent aussi la variable spéciale `$!` en cas d'erreur. Les autres fonctions ne le font pas, sauf de manière accidentelle.

2.1 Fonctions Perl par catégories

Voici les fonctions Perl (y compris ce qui ressemble à des fonctions, comme certains mots-clés et les opérateurs nommés) triées par catégorie. Certaines fonctions apparaissent dans plusieurs catégories à la fois.

Fonctions liées aux scalaires ou aux chaînes de caractères

`chomp`, `chop`, `chr`, `crypt`, `hex`, `index`, `lc`, `lcfirst`, `length`, `oct`, `ord`, `pack`,
`q/CHAINE/`, `qq/CHAINE/`, `reverse`, `rindex`, `sprintf`, `substr`, `tr///`, `uc`, `ucfirst`,
`y///`

Expressions rationnelles et reconnaissances de motifs

`m//`, `pos`, `quotemeta`, `s///`, `split`, `study`, `gr//`

Fonctions numériques

`abs`, `atan2`, `cos`, `exp`, `hex`, `int`, `log`, `oct`, `rand`, `sin`, `sqrt`, `srand`

Fonctions liées aux véritables @tableaux

`pop`, `push`, `shift`, `splice`, `unshift`

Fonctions aux listes de données

`grep`, `join`, `map`, `qw/CHAINE/`, `reverse`, `sort`, `unpack`

Fonctions liées aux véritables %tables de hachage

`delete`, `each`, `exists`, `keys`, `values`

Fonctions d'entrée/sortie

`binmode`, `close`, `closedir`, `dbmclose`, `dbmopen`, `die`, `eof`, `fileno`, `flock`, `format`,
`getc`, `print`, `printf`, `read`, `readdir`, `rewinddir`, `seek`, `seekdir`, `select`, `syscall`,
`sysread`, `sysseek`, `syswrite`, `tell`, `telldir`, `truncate`, `warn`, `write`

Fonctions pour données de longueur fixe ou pour enregistrements

pack, read, syscall, sysread, syswrite, unpack, vec

Fonctions de descripteurs de fichiers, de fichiers ou de répertoires

-X, chdir, chmod, chown, chroot, fcntl, glob, ioctl, link, lstat, mkdir, open, opendir, readlink, rename, rmdir, stat, symlink, sysopen, umask, unlink, utime

Mots-clés liés au contrôle d'exécution de votre programme perl

caller, continue, die, do, dump, eval, exit, goto, last, next, redo, return, sub, wantarray

Mots-clés liés à la portée

caller, import, local, my, our, package, use

Fonctions diverses

defined, dump, eval, formline, local, my, our, reset, scalar, undef, wantarray

Fonctions de gestion des processus et des groupes de processus

alarm, exec, fork, getpgrp, getppid, getpriority, kill, pipe, qx/CHAINE/, setpgrp, setpriority, sleep, system, times, wait, waitpid

Mots-clés liés aux modules perl

do, import, no, package, require, use

Mots-clés liés aux classes et à la programmation orientée objet

bless, dbmclose, dbmopen, package, ref, tie, tied, untie, use

Fonctions de bas niveau liées aux sockets

accept, bind, connect, getpeername, getsockname, getsockopt, listen, recv, send, setsockopt, shutdown, socket, socketpair

Fonctions IPC (communication inter-processus) System V

msgctl, msgget, msgrcv, msgsnd, semctl, semget, semop, shmctl, shmget, shmread, shmwrite

Manipulation des informations sur les utilisateurs et les groupes

endgrent, endhostent, endnetent, endpwent, getgrent, getgrgid, getgrnam, getlogin, getpwent, getpwnam, getpwuid, setgrent, setpwent

Manipulation des informations du réseau

endprotoent, endservent, gethostbyaddr, gethostbyname, gethostent, getnetbyaddr, getnetbyname, getnetent, getprotobyname, getprotobyname, getprotoent, getservbyname, getservbyport, getservent, sethostent, setnetent, setprotoent, setservent

Fonction de date et heure

gmtime, localtime, time, times

Nouvelles fonctions de perl5

perlfunc – Fonctions Perl prédéfinies

abs, bless, chomp, chr, exists, formline, glob, import, lc, lcfirst, map, my, no, our, prototype, qx, qw, readline, readpipe, ref, sub*, sysopen, tie, tied, uc, ucfirst, untie, use

* – sub était un mot-clé dans perl4, mais dans perl5 c'est un opérateur qui peut être utilisé au sein d'expressions.

Fonctions obsolètes en perl5

dbmclose, dbmopen

2.2 Portabilité

Perl est né sur Unix et peut, par conséquent, accéder à tous les appels systèmes Unix courants. Dans des environnements non–Unix, les fonctionnalités de certains appels systèmes Unix peuvent manquer ou différer sur certains détails. Les fonctions Perl affectées par cela sont :

-X, binmode, chmod, chown, chroot, crypt, dbmclose, dbmopen, dump, endgrent, endhostent, endnetent, endprotoent, endpwent, endservent, exec, fcntl, flock, fork, getgrent, getgrgid, gethostent, getlogin, getnetbyaddr, getnetbyname, getnetent, getppid, getparp, getpriority, getprotobynumber, getprotoent, getpwent, getpwnam, getpwuid, getservbyport, getservent, getsockopt, glob, ioctl, kill, link, lstat, msgctl, msgget, msgrcv, msgsnd, open, pipe, readlink, rename, select, semctl, semget, semop, setgrent, sethostent, setnetent, setparp, setpriority, setprotoent, setpwent, setservent, setsockopt, shmctl, shmget, shmread, shmwrite, socket, socketpair, stat, symlink, syscall, sysopen, system, times, truncate, umask, unlink, utime, wait, waitpid

Pour de plus amples détails sur la portabilité de ces fonctions, voir [the perlport manpage](#) et toutes les documentations disponibles spécifiques à la plate–forme considérée.

2.3 Fonctions Perl par ordre alphabétique

–X DESCRIPTEUR

–X EXPR

–X

Un test de fichier où X est une des lettres présentées ci–dessous. Cet opérateur unaire prend soit un argument, soit un nom de fichier, soit un descripteur de fichier, et teste le fichier associé pour constater si quelque chose est vérifié à son sujet. Si l'argument est omis, il teste \$_, sauf –t qui teste STDIN. Sauf indication contraire, il retourne 1 pour VRAI et '' pour FAUX, ou la valeur indéfinie (undef) si le fichier n'existe pas. Malgré leurs noms originaux, leur priorité est la même que celle de tout autre opérateur unaire nommé et l'argument peut–être mis de même entre parenthèses. L'opérateur peut être :

- r Le fichier est en lecture par le uid/gid effectif.
- w Le fichier est en écriture par le uid/gid effectif.
- x Le fichier est exécutable par le uid/gid effectif.
- o Le fichier appartient au uid effectif.
- R Le fichier est en lecture par le uid/gid réel.
- W Le fichier est en écriture par le uid/gid réel.
- X Le fichier est exécutable par le uid/gid réel.
- O Le fichier appartient au uid réel.
- e Le fichier existe.
- z Le fichier a une taille nulle (il est vide).
- s Le fichier n'a pas une taille nulle (retourne sa taille en octets).

perlfunc – Fonctions Perl prédéfinies

- f Le fichier est un fichier normal.
- d Le fichier est un répertoire.
- l Le fichier est un lien symbolique.
- p Le fichier est un tube nommée (FIFO), ou le descripteur est un pipe.
- S Le fichier est une socket.
- b Le fichier est un fichier blocs spécial.
- c Le fichier est un fichier caractères spécial.
- t Le fichier est ouvert sur un tty.
- u Le fichier a le bit setuid positionné.
- g Le fichier a le bit setgid positionné.
- k Le fichier a le sticky bit positionné.
- T Le fichier est un fichier texte ASCII (via une heuristique).
- B Le fichier est un fichier binaire (le contraire de -T).
- M La date de démarrage du script moins la date de dernière modification du fichier (exprimé en jours).
- A Idem pour le dernier accès au fichier.
- C Idem pour le dernier changement de l'inode du fichier (Unix peut avoir un comportement différent des autres systèmes).

Exemple :

```
while (<>) {
    chomp;
    next unless -f $_;      # ignore les fichiers spéciaux
    #...
}
```

L'interprétation des opérateurs de permission sur le fichier `-r`, `-R`, `-w`, `-W`, `-x`, et `-X` est uniquement basée sur le mode du fichier et les uids/gids de l'utilisateur. En fait, il peut y avoir d'autres raisons pour lesquelles vous ne pouvez pas lire, écrire ou exécuter le fichier. Par exemple, le contrôle d'accès aux systèmes de fichiers réseau (NFS), les listes de contrôles d'accès (ACL), les systèmes de fichiers en lecture seule et les formats d'exécutable non reconnus.

Notez aussi que, pour le super-utilisateur, `-r`, `-R`, `-w`, et `-W` retournent toujours 1, et `-x` ainsi que `-X` retournent 1 si l'un des bits d'exécution est positionné dans le mode. Les scripts exécutés par le super-utilisateur peuvent donc nécessiter un appel `stat()` pour déterminer exactement les droits du fichier ou alors effectuer un changement temporaire d'uid.

Si vous utilisez les ACL (listes de contrôles d'accès), il existe un pragma appelé `filetest` qui peut produire des résultats plus précis que les informations minimales des bits de permissions fournies par `stat()`. Lorsque vous faites `use filetest 'access'`, les tests sur fichiers susnommés utiliseront les appels systèmes de la famille `access()`. Notez aussi que dans ce cas, les tests `-x` et `-X` peuvent retourner VRAI même si aucun bit d'exécution n'est positionné (que ce soit les bits normaux ou ceux des ACLs). Ce comportement étrange est dû à la définition des appels systèmes sous-jacents. Lisez la documentation du pragma `filetest` pour de plus amples informations.

Notez que `-s/a/b/` n'effectue pas une substitution négative. Toutefois, écrire `-exp($foo)` fonctionne toujours comme prévu -- seule une lettre isolée après un tiret est interprétée comme un test de fichier.

Les tests `-T` et `-B` fonctionnent de la manière suivante. Le premier bloc du fichier est examiné, à la recherche de caractères spéciaux tels que des codes de contrôle ou des octets avec un bit de poids fort. Si trop de caractères spéciaux (> 30 %) sont rencontrés, c'est un fichier `-B`, sinon c'est un fichier `-T`. De plus, tout fichier contenant un octet nul dans le premier bloc est considéré comme binaire. Si `-T` ou `-B` est utilisé sur un descripteur de fichier, le tampon `stdio` courant est examiné à la place du premier bloc. `-T` et `-B` retournent tous les deux VRAI sur un fichier nul, ou une fin de fichier s'il s'agit d'un descripteur. Comme vous devez lire un fichier pour effectuer le test `-T`, la plupart du temps, vous devriez d'abord utiliser un `-f` sur le fichier, comme dans `next unless -f $file && -T $file`.

Si le descripteur spécial, constitué d'un seul underscore (N.d.T. : caractère souligné), est fourni comme argument d'un test de fichier (ou aux opérateurs `stat()` et `lstat()`), alors c'est la structure stat du dernier fichier traité par un test (ou opérateur) qui est utilisée, épargnant ainsi un appel système. (Ceci ne fonctionne pas avec `-t` et n'oubliez pas que `lstat()` et `-l` laisseront dans la structure stat des informations liées au fichier symbolique et non au fichier réel.) (Notez aussi que si des informations issues d'un appel à `lstat` étaient dans le buffer stat alors `-T` et `-B` les remplaceront par le résultat de `stat _`.)

Exemple :

```
print "Can do.\n" if -r $a || -w _ || -x _;
stat($filename);
print "Readable\n" if -r _;
print "Writable\n" if -w _;
print "Executable\n" if -x _;
print "Setuid\n" if -u _;
print "Setgid\n" if -g _;
print "Sticky\n" if -k _;
print "Text\n" if -T _;
print "Binary\n" if -B _;
```

abs VALEUR

abs

Retourne la valeur absolue de son argument. Si VALEUR est omis, utilise `$_`.

accept NOUVELLESOCKET, GENERIQUESOCKET

Accepte une connexion entrante de socket, tout comme l'appel système `accept(2)`. Retourne l'adresse compacte en cas de succès, FAUX sinon. Voir l'exemple de [Sockets : communication client/serveur in the perlipc manpage](#).

Sur les systèmes qui supportent le drapeau fermeture-à-l-exécution (`close-on-exec`) sur les fichiers, ce drapeau sera positionné pour de nouveaux descripteurs de fichier en fonction de la valeur de `$^F`. Voir [\\$^F in the perlvar manpage](#).

alarm SECONDES

alarm

S'arrange pour qu'un SIGALRM soit délivré au processus après que le nombre spécifié de secondes se soient écoulées. Si SECONDES est omis, la valeur de `$_` est utilisée. (Sur certaines machines, malheureusement, le temps écoulé peut être jusqu'à une seconde de plus ou de moins que celui spécifié, en fonction de la façon dont les secondes sont comptées. De plus, la partage du temps entre les différents processus peut entraîner un retard supplémentaire.)

Il n'est pas possible d'activer plusieurs décomptes temporels à la fois. Chaque appel annule le décompte précédent. La valeur 0 peut être fournie pour annuler le décompte précédent sans en créer un nouveau. La valeur retournée est le temps restant de décompte précédent. Pour des délais d'une précision inférieure à la seconde, vous pouvez utiliser soit la version Perl à quatre paramètres de `select()` en laissant les trois premiers indéfinis soit l'interface `syscall()` de Perl pour accéder à `setitimer(2)` si votre système le supporte. Le module `Time::HiRes` (qui fait partie de Perl depuis la version 5.8 ou disponible sur CPAN sinon) peut aussi s'avérer utile.

C'est souvent une erreur de mélanger des appels à `alarm()` avec des appels à `sleep()` (`sleep` peut-être implémenté via des appels internes à `alarm` sur votre système.)

Si vous souhaitez utiliser `alarm()` pour contrôler la durée d'un appel système, il vous faut utiliser le couple `eval()/die()`. Vous ne pouvez pas compter sur l'alarme qui déclenche l'échec de l'appel système avec `$!` positionné à `EINTR` car Perl met en place des

descripteurs de signaux pour redémarrer ces appels sur certains systèmes. Utiliser [eval\(\)/die\(\)](#) fonctionne toujours sous réserve de l'avertissement signalé dans [Signaux in the perlipc manpage](#).

```
eval {
    local $SIG{ALRM} = sub { die "alarm\n" }; # N.B. : \n obligatoire
    alarm $timeout;
    $nread = sysread SOCKET, $buffer, $size;
    alarm 0;
};
if ($?) {
    die unless $? eq "alarm\n"; # propage des erreurs inattendues
    # délai dépassé : time out
}
else {
    # délai non dépassé
}
```

atan2 Y,X

Retourne l'arc-tangente de Y/X dans l'intervalle $-\pi$ à π .

Pour l'opération tangente, vous pouvez utiliser la fonction POSIX: `tan()` ou la relation habituelle :

```
sub tan { sin($_[0]) / cos($_[0]) }
```

bind SOCKET,NOM

Associe une adresse réseau à une socket, tout comme l'appel système `bind`. Retourne VRAI en cas de succès, FAUX sinon. NOM doit être une adresse compactée (par [pack\(\)](#)) du type approprié pour la socket. Voir les exemples de [Sockets : communication client/serveur in the perlipc manpage](#).

binmode DESCRIPTEUR, FILTRE

binmode DESCRIPTEUR

Positionne DESCRIPTEUR pour être lu ou écrit en mode ``binaire'' ou en mode ``texte'' sur les systèmes d'exploitation qui font la distinction entre fichiers texte et binaire. Si DESCRIPTEUR est une expression, sa valeur est utilisée comme nom du descripteur. Retourne vrai en cas de succès ou [undef](#) en cas d'échec.

Si FILTRE est omis ou s'il vaut `:raw`, le DESCRIPTEUR est positionné pour passer des données binaires. Cela inclut la désactivation d'une éventuelle traduction des CRLF et passe en mode octets (à opposer à des caractères Unicode). Notez bien que, en dépit de ce qui est dit dans ``*Programming Perl*'' (le Camel) ou ailleurs, `:raw` n'est pas l'inverse de `:crlf` -- toute autre filtre (FILTRE ou LAYER en anglais) qui pourrait modifier la nature binaire du flot est aussi désactivée. Voir [PerlIO](#), [the perlrun manpage](#) et tout ce qui est dit de la variable d'environnement PERLIO.

Le paramètre FILTRE de la fonction binmode() est appelé ``DISCIPLINE'' dans ``Programming Perl, 3rd Edition''. Mais, depuis la publication de ce livre connu sous le nom de ``Camel III'', un consensus pour le nommage de ce paramètre est passé de ``discipline'' à ``filtre'' (layer en anglais). Toute la documentation de cette version de Perl se réfère donc maintenant à ``layer'' plutôt qu'à ``discipline''. Revenons maintenant à la documentation...

Sur certains systèmes (en général les systèmes DOS et ceux basés sur Windows), [binmode\(\)](#) est nécessaire lorsque vous ne travaillez pas avec des fichiers texte. Pour assurer la portabilité, c'est une bonne idée de toujours l'utiliser lorsque c'est approprié et de ne jamais l'utiliser dans les autres cas.

En d'autres termes : indépendamment de la plate-forme, utilisez [binmode\(\)](#) sur les fichier binaires et n'utilisez pas [binmode\(\)](#) sur les fichiers textes.

Si `FILTRE` est présent, il doit être une seule chaîne de caractères mais il peut contenir plusieurs directives. Ces directives modifient le comportement du `DESCRIPTEUR`.

L'utilisation de `FILTRE` via `binmode()` sur des fichiers texte a un sens.

Pour marquer un `DESCRIPTEUR` comme UTF-8, utilisez `:utf8`.

Les directives comme `:bytes`, `:crlf`, `:utf8` ou de la forme `:. . .` sont appelées des *filtres I/O* (filtres d'entrée/sortie). La directive `open` peut être utilisée pour spécifier les filtres d'I/O par défaut. Voir `open`.

En général, `binmode()` devrait être appelé après `open()` et avant n'importe quelle opération d'entrée/sortie. L'appel à `binmode()` videra tous les tampons de données en attente de sortie (ou d'entrée). La seule exception est le filtre `:encoding` qui change l'encodage par défaut du `DESCRIPTEUR`. Voir `open`. L'appel au filtre `:encoding` est parfois nécessaire en cours de flot : il ne vide pas les tampons.

Le système d'exploitation, les pilotes de périphériques, les bibliothèques C et l'interpréteur de Perl coopèrent afin de permettre au programmeur de considérer une fin de ligne comme un seul caractère (`\n`) et cela, indépendamment de sa représentation externe. Sur la plupart des systèmes d'exploitation, la représentation utilisée par les fichiers textes natifs correspond à la représentation interne mais sur certaines plates-formes la représentation externe de `\n` est constituée de plus d'un caractère.

Mac OS, toutes les variantes d'UNIX et les `Stream_LF` de VMS utilisent un seul caractère pour représenter une fin de ligne dans leur représentation externe des textes (même si ce caractère unique est un `RETOUR CHARIOT` sur Mac OS et un `SAUT DE LIGNE` sur Unix et dans la plupart des fichiers VMS). Sur d'autres systèmes tels que VMS, MS-DOS et les différentes versions de MS-Windows, votre programme voit un `\n` comme un simple `\cJ` mais ce qui est réellement stocké dans les fichiers textes est le couple de caractères `\cM\cJ`. Cela signifie que, si vous n'utilisez pas `binmode()` sur ces systèmes, les séquences `\cM\cJ` sur disque seront converties en `\n` en entrée et que tous les `\n` produits par votre programme seront reconvertis en `\cM\cJ` à la sortie. C'est ce que vous voulez pour les fichiers textes mais cela peut être désastreux pour un fichier binaire.

Autre conséquence de l'utilisation de `binmode()` (sur certains systèmes) : les marqueurs spéciaux de fin de fichiers seront vus comme faisant partie du flux de données. Pour les systèmes de la famille Microsoft, cela signifie que, si vos données binaires contiennent un `\cZ`, le système d'entrée/sortie le considérera comme une fin de fichier à moins que vous n'utilisiez `binmode()`.

`binmode()` est important non seulement pour les opérations `readline()` et `print()` mais aussi lorsque vous utilisez `read()`, `seek()`, `sysread()`, `syswrite()` et `tell()` (voir [the perlport manpage](#) pour plus d'informations). Voir aussi `$/` et `$\` dans [the perlvar manpage](#) pour savoir comment fixer manuellement les séquences de fin de lignes en entrée et en sortie.

bless REF,NOMCLASSE

bless REF

Cette fonction précise à la chose référencée par `REF` qu'elle est désormais un objet du paquetage `NOMCLASSE` -- ou le paquetage courant si aucun `NOMCLASSE` n'est spécifié, ce qui est souvent le cas. Étant donné que l'appel à `bless()` est souvent la dernière instruction d'un constructeur, cette fonction retourne la référence elle-même. Utilisez toujours la version à deux arguments si la fonction effectuant la "bénédiction" `bless()` peut être héritée par une classe dérivée. Cf. [the perltoot manpage](#) et [the perlobj manpage](#) pour de plus amples informations sur la bénédiction (et les bénédiction) d'objets.

Ne blessez des objets qu'avec des `NOMCLASSEs` mélangeant des majuscules et des minuscules. Les espaces de nommages entièrement en minuscule sont réservés pour les directives (pragmas) Perl. Les types prédéfinis utilisent les noms entièrement en majuscule.

Donc, pour éviter toute confusion, vous devez éviter ces deux types de nommage. Soyez sûr que `NOMCLASSE` est une valeur vraie. Voir [Modules Perl in the perlmod manpage](#).

caller* *EXPR

caller

Retourne le contexte de l'appel de subroutine courant. Dans un contexte scalaire, retourne le nom du paquetage de l'appelant, s'il existe, c'est-à-dire si nous sommes dans une subroutine, un `eval()` ou un `require()`, et retourne la valeur indéfinie (`undef`) sinon. En contexte de liste, retourne

```
($package, $filename, $line) = caller;
```

Avec `EXPR`, il retourne des informations supplémentaires que le débogueur utilise pour afficher un historique de la pile. La valeur de `EXPR` donne le nombre de contextes d'appels à examiner au-dessus de celui en cours.

```
($package, $filename, $line, $subroutine, $hasargs,
 $wantarray, $evaltext, $is_require, $hints, $bitmask) = caller($i);
```

Ici, `$subroutine` peut être `"(eval)"` si le cadre n'est pas un appel de routine mais un `eval()`. Dans un tel cas, les éléments supplémentaires `$evaltext` et `$is_require` sont positionnés : `$is_require` est vrai si le contexte est créé par un `require` ou un `use`, `$evaltext` contient le texte de l'instruction `eval EXPR`. En particulier, pour une instruction `eval BLOC`, `$filename` vaut `"(eval)"` mais `$evaltext` est indéfini. (Notez aussi que chaque instruction `use` crée un contexte `require` à l'intérieur d'un contexte `eval EXPR`.) `$subroutine` peut aussi être `(unknown)` si cette subroutine particulière a disparu de la table des symboles. `$hasargs` est vrai si une nouvelle instance de `@_` a été créé pour ce contexte. Les valeurs de `$hints` et de `$bitmask` risquent de changer d'une version de Perl à une autre. Elles n'ont donc aucun intérêt pour une utilisation externe.

De plus, s'il est appelé depuis le paquetage `DB`, `caller` retourne plus de détails : il affecte à la liste de variables `@DB::args` les arguments avec lesquels la routine a été appelée. Prenez garde à l'optimiseur qui a pu optimiser des contextes d'appel avant que `caller()` ait une chance de récupérer les informations. Ce qui signifie que `caller(N)` pourrait ne pas retourner les informations concernant le contexte d'appel que vous attendez, pour `N > 1`. En particulier, `@DB::args` peut contenir des informations relatives à un appel précédent de `caller()`.

chdir* *EXPR

Change le répertoire courant à `EXPR`, si c'est possible. Si `EXPR` est omis, change vers le répertoire spécifié par la variable `$ENV{HOME}` si elle est définie ; sinon, c'est le répertoire spécifié par la variable `$ENV{LOGDIR}` qui est utilisé. (Sous `VMS`, la variable `$ENV{SYS$LOGIN}` est aussi regardée et utilisée si elle est définie.) Si aucune de ces variables n'est définie, `chdir` seul ne fait rien. Retourne `VRAI` en cas de succès, `FAUX` sinon. Cf. exemple de `die()`.

chmod* *LISTE

Change les permissions d'une liste de fichiers. Le premier élément de la liste doit être le mode numérique, qui devrait être un nombre en octal et *non* une chaîne de chiffres en octal : `0644` est correct, mais pas `'0644'`. Retourne le nombre de fichiers dont les permissions ont été changées avec succès. Voir aussi `oct`, si vous ne disposez que d'une chaîne de chiffres.

```
$cnt = chmod 0755, 'foo', 'bar';
chmod 0755, @executables;
$mode = '0644'; chmod $mode, 'foo';      # !!! fixe le mode à
                                          # --w----r-T
```

perlfunc – Fonctions Perl prédéfinies

```
$mode = '0644'; chmod oct($mode), 'foo'; # ceci est mieux
$mode = 0644;  chmod $mode, 'foo';      # cela est le meilleur
```

Vous pouvez aussi importer les constantes symboliques `S_I*` du module `Fcntl` :

```
use Fcntl ':mode';
chmod S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH|S_IXOTH, @executables;
# Identique au chmod 0755 de l'exemple précédent.
```

chomp VARIABLE

chomp(LISTE)

chomp

Cette version sûre de `chop` supprime toute fin de ligne correspondant à la valeur courante de `$/` (connue aussi sous le nom de `$INPUT_RECORD_SEPARATOR` dans le module `English`). Elle retourne le nombre total de caractères effacés de tous ses arguments. Elle est souvent utilisée pour effacer le saut de ligne de la fin d'une entrée quand vous vous souciez que l'enregistrement final pourrait ne pas avoir ce saut de ligne. En mode paragraphe (`$/ = ""`), elle efface tous les sauts de ligne à la fin de la chaîne de caractères. En mode «slurp» (`$/ = undef`) ou en mode enregistrement de taille fixe (`$/` est une référence vers un entier ou similaire, voir [the perlvar manpage](#)) `chomp()` ne supprime rien du tout. Si `VARIABLE` est omis, elle tronque `$_`. Exemple :

```
while (<>) {
    chomp; # évite le \n du dernier champ
    @array = split(/:/);
    # ...
}
```

Vous pouvez en fait tronquer tout ce qui est une lvalue, y compris les affectations :

```
chomp($cwd = `pwd`);
chomp($answer = <STDIN>);
```

Si vous tronquez une liste, chaque élément est tronqué et le nombre total de caractères effacés est retourné.

Notez que les parenthèses sont nécessaires lorsque vous voulez ``chomp''-er quelque chose qui n'est pas une simple variable. C'est parce que `chomp $cwd = `pwd`` ; est interprété comme `(chomp $cwd) = `pwd`` ; plutôt que comme `chomp($cwd = `pwd`)` ;. De même dans le cas de `chomp $a, $b` qui est interprété comme `chomp($a), $b` plutôt que comme `chomp($a, $b)`.

chop VARIABLE

chop(LISTE)

chop

Efface le dernier caractère d'une chaîne et le retourne. Cet opérateur est utilisé pour effacer le saut de ligne de la fin d'une entrée car il est plus efficace que `s/\n//` étant donné qu'il ne scanne ni ne copie la chaîne. Si `VARIABLE` est omis, tronque `$_`. Exemple :

```
while (<>) {
    chop; # évite \n du dernier champ
    @array = split(/:/);
    #...
}
```

Vous pouvez en fait tronquer tout ce qui est une lvalue, y compris les affectations :

```
chop($cwd = `pwd`);
chop($answer = <STDIN>);
```

Si vous tronquez une liste, chaque élément est tronqué. Seul la valeur du dernier `chop()` est retournée.

Notez que `chop()` retourne le dernier caractère. Pour retourner tout sauf le dernier caractère, utilisez `substr($string, 0, -1)`.

Voir aussi [chomp](#).

chown LISTE

Change le propriétaire (et le groupe) d'une liste de fichiers. Les deux premiers éléments de la liste doivent être, dans l'ordre, les uid et gid *numériques*. Une valeur `-1` à l'une de ces positions est interprétée par la plupart des systèmes comme une valeur à ne pas modifier. Retourne le nombre de fichiers modifiés avec succès.

```
$cnt = chown $uid, $gid, 'foo', 'bar';
chown $uid, $gid, @filenames;
```

Voici un exemple qui cherche les uid non numériques dans le fichier de mots de passe :

```
print "User: ";
chop($user = <STDIN>);
print "Files: ";
chop($pattern = <STDIN>);
($login,$pass,$uid,$gid) = getpwnam($user)
    or die "$user not in passwd file";
@ary = glob($pattern);      # expansion des noms de fichiers
chown $uid, $gid, @ary;
```

Sur la plupart des systèmes, vous n'êtes pas autorisé à changer le propriétaire d'un fichier à moins d'être le super-utilisateur, même si avez la possibilité de changer un groupe en l'un de vos groupes secondaires. Sur les systèmes non sécurisés, ces restrictions peuvent être moindres, mais ceci n'est pas une hypothèse portable. Sur les systèmes POSIX, vous pouvez détecter cette condition de la manière suivante :

```
use POSIX qw(sysconf _PC_CHOWN_RESTRICTED);
$can_chown_giveaway = not sysconf(_PC_CHOWN_RESTRICTED);
```

chr NOMBRE

chr

Retourne le caractère représenté par ce NOMBRE dans le jeu de caractères. Par exemple, `chr(65)` est "A" en ASCII ou Unicode et `chr(0x263a)` est un visage réjoui en Unicode. Notez que les caractères de 127 à 255 (inclu) ne sont pas par défaut encodés en Unicode pour des raisons de compatibilités (voir *encoding*).

Pour la fonction réciproque, utilisez `ord`. Voir *perlunicode* et *encoding* pour en savoir plus sur Unicode.

Si NOMBRE est omis, utilise `$_`.

chroot FICHER

chroot

Cet opérateur fonctionne comme l'appel système du même nom : il fait du répertoire spécifié le nouveau répertoire racine pour tous les chemins commençant par un "/" utilisés par votre processus et ses enfants. (Ceci n'affecte pas le répertoire courant qui reste inchangé.) Pour des raisons de sécurité, cet appel est restreint au super-utilisateur. Si FICHER est omis, effectue un `chroot()` sur `$_`.

close DESCRIPTEUR

close

Ferme le fichier ou le tube associé au descripteur de fichier, en retournant VRAI si et seulement si les tampons d'entrée/sortie ont été correctement vidés et si le descripteur a été correctement fermé. Ferme le descripteur courant si l'argument est omis.

Vous n'avez pas à fermer le DESCRIPTEUR si vous allez immédiatement refaire un `open()` sur celui-ci, car `open()` le fermera pour vous. (voir `open()`.) Toutefois, un `close()` explicite sur un fichier d'entrée réinitialise le compteur de lignes (`$.`) alors que la fermeture implicite par un `open()` ne le fait pas.

Si le descripteur vient d'un tube ouvert, `close()` va de plus retourner FAUX si un des autres appels système impliqués échoue ou si le programme se termine avec un statut non

nul. (Si le seul problème rencontré est une terminaison de programme non nulle, \$! sera à 0.) De même, la fermeture d'un tube attend que le programme exécuté sur le tube soit achevé, au cas où vous souhaiteriez voir la sortie du tube après coup, et positionne implicitement la valeur du statut de sortie de la commande dans \$?.

La fermeture prématurée d'une extrémité de lecture d'un tube (c'est-à-dire avant que le processus qui écrit à l'autre extrémité l'ait fermé) aura pour conséquence l'envoi d'un signal SIGPIPE au processus écrivain. Si l'autre extrémité n'est pas prévue pour gérer cela, soyez sûr d'avoir lu toutes les données avant de fermer le tube.

Exemple :

```
open(OUTPUT, '|sort >foo') # tube vers sort
  or die "Can't start sort: $!";
#...                               # imprime des trucs sur la sortie
close OUTPUT                       # attend la fin de sort
  or warn $! ? "Error closing sort pipe: $!"
    : "Exit status $? from sort";
open(INPUT, 'foo')                 # recupere les resultats de sort
  or die "Can't open 'foo' for input: $!";
```

Le DESCRIPTEUR peut être une expression dont la valeur peut être utilisée en tant que descripteur indirect, habituellement le véritable nom du descripteur.

***closedir* REPDESCRIPTEUR**

Ferme un répertoire ouvert par opendir() et retourne le résultat de cet appel système. Le REPDESCRIPTEUR peut être une expression dont la valeur peut être utilisée comme un descripteur indirect de répertoire, habituellement le véritable nom du descripteur de répertoire.

***connect* SOCKET,NOM**

Tente une connexion sur une socket distante, tout comme le fait l'appel système du même nom. Retourne VRAI en cas de succès, FAUX sinon. Le NOM doit être une adresse compactée (par pack()) du type approprié correspondant à la socket. Voir les exemples de Sockets : communication Client/Serveur in the perlipc manpage.

***continue* BLOC**

En fait, c'est une instruction de contrôle d'exécution plutôt qu'une fonction. S'il y a un BLOC continue rattaché à un BLOC (typiquement dans un `while` ou un `foreach`), il est toujours exécuté juste avant le test à évaluer à nouveau, tout comme la troisième partie d'une boucle `for` en C. Il peut donc être utilisé pour incrémenter une variable de boucle, même si la boucle a été continuée par l'instruction next (qui est similaire à l'instruction continue en C).

last, next, ou redo peuvent apparaître dans un bloc continue. last et redo vont se comporter comme s'ils avaient été exécutés dans le bloc principal. De même pour next, mais comme il va exécuter un bloc continue, il peut s'avérer encore plus divertissant.

```
while (EXPR) {
    ### redo vient toujours ici
    do_something;
} continue {
    ### next vient toujours ici
    do_something_else;
    # puis retour au sommet pour revérifier EXPR
}
### last vient toujours ici
```

Omettre la section continue est sémantiquement équivalent à en utiliser une vide, de manière assez logique. Dans ce cas, next retourne directement vérifier la condition au sommet de la boucle.

cos *EXPR*

Retourne le cosinus de *EXPR* (exprimé en radians). Si *EXPR* est omis, calcule le cosinus de `$_`.

Pour la fonction réciproque, vous pouvez utiliser la fonction `Math::Trig::acos()` ou alors utiliser cette relation :

```
sub acos { atan2( sqrt(1 - $_[0] * $_[0]), $_[0] ) }
```

crypt *TEXTE,SEL*

Crypte une chaîne de caractères exactement comme la fonction `crypt(3)` de la bibliothèque C (en supposant que vous en avez une version dont elle n'a pas été extirpée en tant que munition potentielle). Ceci peut s'avérer utile, entre autres choses, pour vérifier le fichier de mots de passe à la recherche de mots de passe bancals. Seuls les types portant un chapeau blanc devraient faire ça.

Notez que `crypt()` est conçu pour être une fonction à sens unique, un peu comme les oeufs qu'on casse pour faire une omelette. Il n'y a aucune fonction (connue) pour décrypter (donc `crypt()` est une fonction de hachage à sens unique). Par conséquent, cette fonction n'est pas du tout utile pour de la cryptographie. (Pour ça, voyez plutôt votre miroir CPAN le plus proche.)

Pour vérifier une chaîne cryptée, vous devriez utiliser cette chaîne cryptée comme *SEL* (comme dans `crypt($plain, $crypté) eq $crypté`). Cela permet à votre script de fonctionner aussi bien avec le `crypt` standard qu'avec des implémentations plus exotiques. En d'autres termes, vous ne pouvez rien supposer sur le contenu de la chaîne produite ou sur la taille de la chaîne ainsi chiffrée.

Traditionnellement, le résultat est une chaîne de 13 octets : les deux premiers octets du *SEL*, suivis de 11 octets pris dans l'ensemble `./0-9A-Za-z` et seul les huit premiers octets de la chaîne à encrypter comptent. Mais des méthodes de hachages alternatives (comme MD5) ou un niveau de sécurité plus haut (comme C2) ou des implémentations sur des plateformes non-UNIX peuvent produire d'autres types de chaînes.

Lorsque vous choisissez un nouveau *SEL* constitué de deux caractères aléatoires, ces caractères doivent provenir de l'ensemble `./0-9A-Za-z` (Exemple `join ' ', ('.', '/', 0..9, 'A'..'Z', 'a'..'z')[rand 64, rand 64]`).

Voici un exemple qui garantit que quiconque lance ce programme connaît son propre mot de passe :

```
$pwd = (getpwuid($<))[1];
$salt = substr($pwd, 0, 2);
system "stty -echo";
print "Password: ";
chop($word = <STDIN>);
print "\n";
system "stty echo";
if (crypt($word, $salt) ne $pwd) {
    die "Sorry...\n";
} else {
    print "ok\n";
}
```

Bien évidemment, donner votre mot de passe à quiconque vous le demande est très peu recommandé.

La fonction `crypt` n'est pas utilisable pour chiffrer de grande quantité d'information déjà parce que vous ne pouvez pas retrouver l'information initiale. Regardez les répertoires *by-module/Crypt* et *by-module/PGP* sur votre miroir CPAN préféré pour y trouver des modules plus pratiques.

Si vous utilisez `crypt()` sur une chaîne Unicode (qui contient *éventuellement* des caractères dont l'encodage est supérieur à 255), Perl essaie de se sortir de cette situation

en dégradant une copie de la chaîne vers un encodage sur 8 bits avant d'appeler `crypt()` sur cette copie. Si cela fonctionne, tout est bon. Sinon, `crypt()` meurt (die) avec les message `Wide character in crypt`.

dbmclose HASH

[Cette fonction a été remplacée par la fonction `untie()`.]
Rompt le lien entre un fichier DBM et une table de hachage.

dbmopen HASH,DBNOM,MODE

[Cette fonction a été remplacée par la fonction `tie()`.]

Cette fonction lie un fichier `dbm(3)`, `ndbm(3)`, `sdbm(3)`, `gdbm(3)`, ou Berkeley DB à une table de hachage. `HASH` est le nom de la table de hachage. (À la différence du `open()` normal, le premier argument n'est *pas* un descripteur de fichier, même s'il en a l'air). `DBNOM` est le nom de la base de données (sans l'extension `.dir` ou `.pag`, le cas échéant). Si la base de données n'existe pas, elle est créée avec les droits spécifiés par `MODE` (et modifiés par `umask`). Si votre système supporte uniquement les anciennes fonctions DBM, vous ne pouvez exécuter qu'un seul `dbmopen()` dans votre programme. Dans les anciennes versions de Perl, si votre système n'avait ni DBM ni ndbm, l'appel à `dbmopen()` produisait une erreur fatale ; il utilise maintenant `sdbm(3)`.

Si vous n'avez pas les droits d'écriture sur le fichier DBM, vous pouvez seulement lire les variables de la table de hachage, vous ne pouvez pas y écrire. Si vous souhaitez tester si vous pouvez y écrire, faites un test sur le fichier ou essayez d'écrire une entrée bidon dans la table de hachage, à l'intérieur d'un `eval()`, qui interceptera l'erreur.

Notez que les fonctions telles que `keys()` et `values()` peuvent retourner des listes gigantesques si elles sont utilisées sur de gros fichiers DBM. Vous devriez préférer la fonction `each()` pour parcourir des fichiers DBM volumineux. Exemple :

```
# imprime en sortie les index du fichier d'historiques
dbmopen(%HIST, '/usr/lib/news/history', 0666);
while (($key,$val) = each %HIST) {
    print $key, ' = ', unpack('L',$val), "\n";
}
dbmclose(%HIST);
```

Voir aussi *AnyDBM_File* pour une description plus générale des avantages et inconvénients des différentes approches `dbm` ainsi que *DB_File* pour une implémentation particulièrement riche.

Vous pouvez contrôler la bibliothèque DBM utilisé en chargeant cette bibliothèque avant l'appel à `dbmopen()` :

```
use DB_File;
dbmopen(%NS_Hist, "$ENV{HOME}/.netscape/history.db")
or die "Can't open netscape history file: $!";
```

defined EXPR

defined

Retourne une valeur booléenne exprimant si `EXPR` a une valeur autre que la valeur indéfinie `undef`. Si `EXPR` est absent, `$_` sera vérifiée de la sorte.

De nombreuses opérations retournent `undef` pour signaler un échec, la fin d'un fichier, une erreur système, une variable non initialisée et d'autres conditions exceptionnelles. Cette fonction vous permet de distinguer `undef` des autres valeurs. (Un simple test booléen ne distinguera pas `undef`, zéro, la chaîne de caractères vide et `<`0">`, qui représentent tous faux.) Notez que puisque `undef` est un scalaire valide, sa présence n'indique pas *nécessairement* une condition exceptionnelle : `pop()` retourne `undef` quand son argument est un tableau vide *ou* quand l'élément à retourner a la valeur `undef`.

Vous pouvez aussi utiliser `defined(&func)` pour vérifier si la subroutine `&func` a déjà été définie. La valeur retournée ne sera pas affectée par une déclaration préalable de `&func`. L'utilisation de `defined()` sur des agrégats (tables de hachage et tableaux) est dépréciée. C'était utilisé pour savoir si de la mémoire avait déjà été allouée pour cet agrégat. Ce comportement pourrait disparaître dans une future version de Perl. Vous devriez utiliser un simple test de taille à la place :

```
if (@an_array) { print "has array elements\n" }
if (%a_hash)   { print "has hash members\n" }
```

Utilisé sur un élément de table de hachage, il vous indique si la valeur est définie, mais pas si la clé existe dans la table. Utilisez `exists` dans ce but.

Exemples :

```
print if defined $switch{'D'};
print "$val\n" while defined($val = pop(@ary));
die "Can't readlink $sym: $!"
    unless defined($value = readlink $sym);
sub foo { defined &$bar ? &$bar(@_) : die "No bar"; }
$debugging = 0 unless defined $debugging;
```

Note : de nombreuses personnes ont tendance à trop utiliser `defined()` et sont donc surprises de découvrir que le nombre 0 et " " (la chaîne de caractères vide) sont, en fait, des valeurs définies. Par exemple, si vous écrivez :

```
"ab" =~ /a(.*?)b/;
```

La recherche de motif réussit et `$1` est définie, malgré le fait qu'il ne correspond à ``rien``. Mais elle n'a pas véritablement rien trouvé -- elle a plutôt trouvé quelque chose qui s'est avéré être d'une longueur de 0 caractères. Tout ceci reste très loyal et honnête. Quand une fonction retourne une valeur indéfinie, il est admis qu'elle ne pourrait pas vous donner une réponse honnête. Vous devriez donc utiliser `defined()` uniquement lorsque vous vous posez des questions sur l'intégrité de ce que vous tentez de faire. Sinon, une simple comparaison avec 0 ou " " est ce que vous voulez.

Voir aussi `undef`, `exists`, `ref`.

***delete* EXPR**

À partir d'une expression EXPR qui spécifie un élément ou un ensemble d'éléments (slice) d'une table de hachage ou d'un tableau, supprime `1e(s)` élément(s) spécifié(s) de la table de hachage ou du tableau. Dans le cas d'un tableau, si les éléments supprimés se trouvent à la fin du tableau, la taille du tableau est réduite à l'indice le plus grand de tous les indices des éléments restants qui donnent une valeur vraie lorsqu'on teste leur existence via `exists()` (ou 0 si aucun élément n'existe).

Retourne tous les éléments ainsi supprimés ou la valeur indéfinie (`undef`) si aucun élément n'a été supprimé. Supprimer des éléments dans `$ENV{}` modifie les variables d'environnement. Supprimer des éléments d'une table de hachage liée à un fichier DBM supprime ces éléments du fichier. Supprimer des éléments d'une table de hachage ou d'un tableau lié ne retourne pas nécessairement quelque chose.

Supprimer un élément d'un tableau réinitialise effectivement cet emplacement à sa valeur indéfinie initiale. En conséquence, un test d'existence sur cet élément via `exists()` retournera faux. Notez bien que supprimer des éléments au milieu d'un tableau ne décale pas vers le bas les indices des éléments suivants -- utiliser `splice()` pour cela. Voir `exists`.

Le code suivant supprime (de manière inefficace) toutes les valeurs de %HASH et de @TABLEAU :

```
foreach $key (keys %HASH) {
    delete $HASH{$key};
}
foreach $index (0 .. $#TABLEAU) {
```

perlfunc – Fonctions Perl prédéfinies

```
delete $TABLEAU[$index];
}
```

De même que le code suivant :

```
delete @HASH{keys %HASH}
delete @TABLEAU[0 .. $#TABLEAU];
```

Ces deux méthodes restent toutefois beaucoup plus lentes que la simple assignation de la liste vide ou l'utilisation de `undef()` sur `%HASH` ou `@TABLEAU` :

```
%HASH = ();          # vider complètement %HASH
undef %HASH;         # oublier que %HASH a existé
@TABLEAU = ();       # vider complètement @TABLEAU
undef @TABLEAU;     # oublier que @TABLEAU a existé
```

Notez que `EXPR` peut être arbitrairement compliquée tant que l'opération finale se réfère à un élément ou une partie d'une table de hachage ou d'un tableau :

```
delete $ref->[$x][$y]{$key};
delete @{$ref->[$x][$y]}{$key1, $key2, @morekeys};
delete $ref->[$x][$y][$index];
delete @{$ref->[$x][$y]}[$index1, $index2, @moreindices];
```

die LISTE

En dehors d'un `eval()`, affiche les valeur de la LISTE sur `STDERR` et quitte avec la valeur actuelle de `$!` (numéro d'erreur `errno`). Si `$!` est 0, sort avec la valeur (`$? >> 8`) (statut d'une `commande` entre simples apostrophes inverses). Si (`$? >> 8`) est 0, sort avec 255. À l'intérieur d'un `eval()`, le message d'erreur est mis dans `$@` et le `eval()` s'achève sur la valeur indéfinie. Ce qui fait de `die()` la façon de soulever une exception.

Exemples équivalents :

```
die "Can't cd to spool: $!\n" unless chdir '/usr/spool/news';
chdir '/usr/spool/news' or die "Can't cd to spool: $!\n"
```

Si la dernière valeur de LISTE ne se termine pas par un saut de ligne, le numéro de la ligne actuelle du script et le numéro de la ligne d'entrée (le cas échéant) sont aussi affichés et un saut de ligne est ajouté. Notez que ``le numéro de la ligne d'entrée" (mieux connu sous le nom de ``chunk") est dépendant de la notion de ``ligne" courante et est disponible dans la variable spéciale `$.` Voir [\\$/ in the perlvar manpage](#) et [\\$. in the perlvar manpage](#).

Astuce : parfois, la concaténation de `", stopped"` à votre message le rendra plus sensé lorsque la chaîne de caractères `"at foo line 123"` sera ajoutée. Supposez que vous exécutez le script `canasta`.

```
die "/etc/games is no good";
die "/etc/games is no good, stopped";
```

va respectivement produire

```
/etc/games is no good at canasta line 123.
/etc/games is no good, stopped at canasta line 123.
```

Voir aussi `exit()` et `warn()` et le module `Carp`.

Si la LISTE est vide et que `$@` contient déjà une valeur (typiquement provenant d'une évaluation précédente), cette valeur est réutilisée après la concaténation de `"\t...propagated"`. Ceci est utile pour propager des exceptions :

```
eval { ... };
die unless $@ =~ /Expected exception/;
```

Si LISTE est vide et que `$@` contient une référence vers un objet qui a une méthode `PROPAGATE`, cette méthode sera appelée avec comme argument le nom du fichier et le numéro de ligne. La valeur retournée remplacera la valeur dans `$@` (comme si `<$@ = eval { $@-PROPAGATE(__FILE__, __LINE__) };>>` était appelé).

Si `$@` est vide, alors la chaîne de caractères `"Died"` est utilisée.

L'argument de `die()` peut aussi être une référence. Lorsque cela arrive à l'intérieur d'un `eval()` alors `$@` contiendra cette référence. Ce comportement autorise une implémentation plus élaborée de la gestion des exceptions utilisant des objets contenant

une description arbitraire de la nature de l'exception. Une telle utilisation est parfois préférable à la reconnaissance d'un motif particulier par une expression rationnelle appliquée la valeur de \$@. Voici un exemple :

```
eval { ... ; die Some::Module::Exception->new( FOO => "bar" ) };
if ($@) {
    if (ref($@) && UNIVERSAL::isa($@, "Some::Module::Exception")) {
        # gestion de Some::Module::Exception
    }
    else {
        # gestion de toutes les autres exceptions
    }
}
```

Étant donné que perl transformera en chaîne toutes les exceptions non captées avant de les afficher, vous voudrez peut-être surcharger l'opération de transformation en chaîne de tous vos objets représentant des exceptions. Regardez *overload* pour faire cela.

Vous pouvez vous arranger pour qu'une subroutine (de callback) soit appelée juste après le `die()` en l'attachant à `$_SIG{__DIE__}`. La subroutine associée sera appelée avec le texte de l'erreur et peut changer le message de l'erreur, le cas échéant, en appelant `die()` à nouveau. Voir `$_SIG{expr} in the perlvar manpage` pour les détails sur l'assignation des entrées de `%SIG` et `eval BLOC` pour des exemples. Bien que cette fonctionnalité ait été conçue afin d'être utilisée juste au moment où votre script se termine, ce n'est pas le cas -- la subroutine attachée à `$_SIG{__DIE__}` peut aussi être appelée lors de l'évaluation d'un bloc ou d'une chaîne ! Si vous voulez que votre subroutine ne fasse rien dans ce cas, utilisez :

```
die @_ if $^S;
```

comme première ligne de votre subroutine (Voir `$^S in the perlvar manpage`). Étant donné que cela peut engendrer des choses étranges, ce comportement contre-intuitif devrait être changé dans une prochaine version.

do BLOC

Pas vraiment une fonction. Retourne la valeur de la dernière commande dans la suite de commandes contenues dans BLOC. Lorsque modifié par un modificateur de boucle, exécute le BLOC une fois avant le test de la condition de boucle. (Dans les autres cas, le modificateur de boucle teste la condition d'abord.)

`do BLOC` n'est pas considéré comme une boucle et donc les instructions de contrôle de boucle `next`, `last` et `redo` ne peuvent pas être utilisées pour quitter ou redémarrer le bloc. Voir `the perlsyn manpage` pour des stratégies alternatives.

do ROUTINE(LISTE)

Forme dépréciée d'appel de routine. Voir `the perlsub manpage`.

do EXPR

Utilise la valeur de EXPR comme nom de fichier et exécute le contenu de ce fichier en tant que script Perl. Son utilité première est d'inclure les subroutines d'une bibliothèque de subroutines Perl.

```
do 'stat.pl';
```

est identique à

```
eval `cat stat.pl`;
```

sauf que c'est plus efficace et concis, qu'une trace du fichier courant est gardée pour les messages d'erreur, que la recherche du fichier a lieu dans tous les répertoires de `@INC` et que, si le fichier est trouvé, `%INC` est mis à jour. Voir `Noms prédéfinis in the perlvar manpage` pour ces variables. De plus, les variables lexicales visibles lors de l'appel ne sont pas vues par `do NOMFICHER` alors qu'elles le sont par `eval CHAINE`. En revanche, dans

les deux cas, le fichier est retraité à chaque fois que vous l'appeler ce qui n'est probablement pas ce que vous voudriez faire à l'intérieur d'une boucle.

Si `do` ne peut pas lire le fichier, il retourne undef et assigne l'erreur à `$!`. Si `do` peut lire le fichier mais non le compiler, il retourne undef et assigne un message d'erreur à `$@`. Si le fichier est compilé avec succès, `do` retourne la valeur de la dernière expression évaluée. Notez que l'inclusion de modules de bibliothèques est mieux faite par les opérateurs `use()` et `require()` qui effectuent aussi une vérification automatique des erreurs et soulèvent une exception s'il y a le moindre problème.

Vous pourriez aimer utiliser `do` pour lire le fichier de configuration d'un programme. La vérification manuelle d'erreurs peut être effectuée de la façon suivante :

```
# lecture des fichiers : d'abord le système puis l'utilisateur
for $file ("/share/prog/defaults.rc",
           "$ENV{HOME}/.someprogrc") {
    unless ($return = do $file) {
        warn "couldn't parse $file: @$" if $@;
        warn "couldn't do $file: $!"      unless defined $return;
        warn "couldn't run $file"        unless $return;
    }
}
```

dump LABEL

Ceci provoque immédiatement la création de l'image mémoire du programme (core dump). Voir aussi l'option de ligne de commande `-u` dans [the perlrun manpage](#) qui fait la même chose. Au départ, ceci permet d'utiliser le programme `undump` (non fourni) pour convertir votre image mémoire en un programme binaire exécutable après avoir initialisé toutes vos variables au début du programme. Quand le nouveau binaire est exécuté, il va commencer par exécuter un `goto LABEL` (avec toutes les restrictions dont souffre `goto`). Pensez-y comme un branchement goto avec l'intervention d'une image mémoire et une réincarnation. Si LABEL est omis, relance le programme au début.

ATTENTION : tout fichier ouvert au moment de la copie ne sera PAS ouvert à nouveau quand le programme sera réincarné, avec pour résultat une confusion possible sur la portion de Perl.

Cet opérateur est très largement obsolète, en partie parce qu'il est très difficile de convertir un fichier d'image mémoire (core) en exécutable mais aussi parce que le véritable compilateur perl-en-C l'a surpassé. C'est pourquoi vous devez maintenant l'appeler par `CORE:::dump()` si vous ne voulez pas recevoir un message d'avertissement au sujet d'une erreur de frappe possible.

Si vous projetez d'utiliser `dump` pour augmenter la vitesse de votre programme, essayez donc de produire du pseudo-code (ou bytecode) ou du C natif comme cela est décrit dans *perlcc*. Si vous essayez juste d'accélérer un script CGI, regardez donc du côté de l'extension `mod_perl` de **Apache** ou de celui du module `CGI::Fast` sur CPAN. Vous pouvez aussi envisager l'autochargement (autoloading ou selfloading) qui donnera au moins la sensation que votre programme va plus vite.

each HASH

Appelé dans un contexte de liste, retourne une liste de deux éléments constituée de la clé et de la valeur du prochain élément d'une table de hachage, de sorte que vous puissiez itérer sur celle-ci. Appelé dans un contexte scalaire, retourne uniquement la clé de l'élément suivant de la table de hachage.

Les entrées sont apparemment retournées dans un ordre aléatoire. Cet ordre aléatoire réel pourrait changer dans les versions futures de perl mais vous avez la garantie que cet ordre sera le même, que vous utilisiez la fonction `keys` ou la fonction `values` sur une même table

de hachage (non modifiée).

Quand la table de hachage est entièrement lue, un tableau nul est retourné dans un contexte de liste (qui, lorsqu'il est assigné, produit une valeur FAUSSE 0), et undef dans un contexte scalaire. Le prochain appel à each après cela redémarrera une nouvelle itération. Il y a un seul itérateur pour chaque table de hachage, partagé par tous les appels à each, keys ou values du programme ; il peut être réinitialisé en lisant tous les éléments de la table ou en évaluant keys HASH ou values HASH. Si vous ajoutez ou supprimez des éléments d'une table de hachage pendant que vous itérez sur celle-ci, vous pourriez avoir des entrées ignorées ou dupliquées, donc ne le faites pas.

Le code suivant affiche votre environnement comme le fait le programme `printenv(1)`, mais dans un ordre différent :

```
while (($key,$value) = each %ENV) {
    print "$key=$value\n";
}
```

Voir aussi keys() et values().

eof DESCRIPTEUR

eof ()

eof

Retourne 1 si la prochaine lecture sur un DESCRIPTEUR de fichier va retourner une fin de fichier, ou si le DESCRIPTEUR n'est pas ouvert. Le DESCRIPTEUR peut être une expression dont la valeur donne un véritable descripteur de fichier. (Notez que cette fonction lit en fait un caractère puis le retire comme `ungetc()`, elle n'est donc pas vraiment utile dans un contexte interactif.) Ne faites pas de lecture sur un fichier d'un terminal (ou d'appel à eof(DESCRIPTEUR) sur celui-ci) après qu'une fin de fichier soit atteinte. Les types de fichiers comme les terminaux peuvent perdre la condition de fin de fichier si vous le faites. Un eof sans un argument utilise le dernier fichier lu comme argument. Utiliser eof() avec des parenthèses vides est très différent. Il se réfère alors au pseudo fichier formé par les fichiers listés sur la ligne de commande et lu par l'opérateur `<>`. Puisque `<>` n'est pas explicitement ouvert comme l'est un descripteur de fichier normal, l'utilisation de eof() avant celle de `<>` déclenchera l'examen de `@ARGV` pour voir si une entrée est disponible. De manière similaire, un eof() après que `<>` a retourné la condition fin-de-fichier supposera que vous traitez un autre fichier de la liste `@ARGV` ou que vous lisez depuis `STDIN`. Voir Les opérateurs d'E/S in the perlop manpage.

Dans une boucle `while (<>)` eof(ARGV) ou eof peuvent être utilisés pour tester la fin de **CHAQUE** fichier alors que eof() ne détectera que la fin du tout dernier fichier. Exemple :

```
# réinitialise le numérotage des lignes sur chaque fichier d'entrée
while (<>) {
    next if /^\\s*#/;          # saute les commentaires
    print "$\\.\\t$\\_";
} continue {
    close ARGV if eof;       # ce n'est pas eof() !
}
# insère des tirets juste avant la dernière ligne du dernier fichier
while (<>) {
    if (eof()) {             # vérifie la fin du fichier courant
        print "-----\\n";
        close(ARGV);        # ferme ou s'arrête ; nécessaire en cas
                             # de lecture sur le terminal
    }
    print;
}
```

Astuce pratique : vous n'avez presque jamais besoin d'utiliser eof en Perl parce que les opérateurs d'entrée retournent la valeur undef quand ils n'ont plus d'informations à lire ou

s'il y a eu une erreur.

eval EXPR

eval BLOC

Dans sa première forme, la valeur retournée par EXPR est parcourue puis exécutée comme un petit programme Perl. La valeur de l'expression (qui est elle-même déterminée dans un contexte scalaire) est d'abord parcourue puis, s'il n'y a pas eu d'erreurs, exécutée dans le contexte du programme courant, de telle sorte que toute assignation de variable, et toute définition de sous-routine ou de format perdure après cette exécution. Notez que la valeur est parcourue à chaque exécution d'évaluation. Si EXPR est omis, évalué `$_`. Cette forme est typiquement utilisée pour repousser la compilation et l'exécution du texte contenu dans EXPR jusqu'à l'exécution du programme.

Dans sa seconde forme, le code à l'intérieur du BLOC est parcouru une seule fois -- au même moment que la compilation du code entourant l'évaluation -- et exécuté dans le contexte du programme Perl courant. Cette forme est typiquement utilisée pour intercepter des exceptions plus efficacement que la première (voir ci-dessous), en fournissant aussi le bénéfice d'une vérification du code dans le BLOC lors de la compilation.

Le point-virgule final, le cas échéant, peut être omis dans EXPR ou à l'intérieur du BLOC. Dans les deux formes, la valeur retournée est la valeur de la dernière expression évaluée à l'intérieur du mini-programme ; il est aussi possible d'utiliser un retour explicite (via `return`), exactement comme pour les routines. L'expression fournissant la valeur de retour est évaluée en contexte vide, scalaire ou de liste, en fonction du contexte de l'évaluation elle-même. Voir [wantarray](#) pour de plus amples informations sur la façon dont le contexte d'évaluation peut être déterminé.

En cas d'erreur de syntaxe ou d'exécution ou si une instruction `die()` est exécutée, une valeur indéfinie (`undef`) est retournée par `eval()` et le message d'erreur est assigné à `$@`. S'il n'y a pas d'erreur, vous avez la garantie que la valeur de `$@` sera une chaîne de caractères vide. Prenez garde au fait qu'utiliser `eval()` ne dispense Perl ni d'afficher des messages d'alerte (`warnings`) sur `STDERR` ni d'assigner ses messages d'alerte dans `$@`. Pour ce faire, il vous faut utiliser les capacités de `$_SIG{__WARN__}` ou désactiver les messages d'avertissement dans le BLOC ou l'EXPR en utilisant `no warnings 'all'`. Voir [warn](#), [the perlvar manpage](#), `warnings` et `perllexwarn`.

Étant donné que `eval()` intercepte les erreurs non fatales, c'est très pratique pour déterminer si une fonctionnalité (telle que `socket()` ou `symlink()`) est supportée. C'est aussi le mécanisme d'interception d'exception de Perl lorsque l'opérateur `die` est utilisé pour les soulever.

Si le code à exécuter ne varie pas, vous devriez utiliser la seconde forme avec un BLOC pour intercepter les erreurs d'exécution sans supporter l'inconvénient de le recompiler à chaque fois. L'erreur, le cas échéant, est toujours retournée dans `$@`. Exemples :

```
# rend une division par zéro non fatale
eval { $answer = $a / $b; }; warn $@ if $@;
# même chose, mais moins efficace
eval '$answer = $a / $b'; warn $@ if $@;
# une erreur de compilation
eval { $answer = }; # MAUVAIS
# une erreur d'exécution
eval '$answer ='; # sets $@
```

En utilisant la forme `eval{}` pour une interception d'exception dans des bibliothèques, vous pourriez souhaiter ne pas exécuter une éventuelle sous-routine attachée à `__DIE__` par le code de l'utilisateur. Dans ce cas, vous pouvez utiliser la construction `local`

`$_SIG{__DIE__}` comme dans l'exemple ci-dessous :

```
# une interception d'exception de division par zéro très privée
eval { local $_SIG{'__DIE__'}; $answer = $a / $b; };
```

```
warn $@ if $@;
```

Ceci est spécialement significatif, étant donné que les subroutine attachée à `__DIE__` peuvent appeler `die()` à nouveau, ce qui a pour effet de changer leurs messages d'erreur :

```
# les subroutine attachées à __DIE__
# peuvent modifier les messages d'erreur
{
    local $SIG{'__DIE__'} =
        sub { (my $x = $_[0]) =~ s/foo/bar/g; die $x };
    eval { die "foo lives here" };
    print $@ if $@;           # affiche "bar lives here"
}
```

Étant donné que cela favorise une action à distance, ce comportement contre-intuitif pourrait changer dans une version future.

Avec `eval()`, vous devriez faire particulièrement attention à ce qui est examiné, et quand :

```
eval $x;           # CAS 1
eval "$x";        # CAS 2
eval '$x';        # CAS 3
eval { $x };      # CAS 4
eval "\$$x++";    # CAS 5
$$x++;           # CAS 6
```

Les cas 1 et 2 ci-dessus se comportent de la même façon : ils exécutent le code inclus dans la variable `$x`. (Bien que le cas 2 ait des guillemets qui font se demander au lecteur ce qui se passe — réponse : rien.) Les cas 3 et 4 se comportent aussi de la même façon : ils exécutent le code `'$x'` qui ne fait rien que retourner la valeur de `$x`. (Le cas 4 est préférable pour des raisons purement visuelles mais aussi parce qu'il a l'avantage d'être compilé lors de la compilation plutôt que lors de l'exécution.) Le cas 5 est un endroit où vous *DEVRIEZ* normalement souhaiter utiliser des guillemets, sauf que dans ce cas particulier, vous pouvez juste utiliser les références symboliques à la place, comme dans le cas 6. Un `eval BLOC` n'est pas considéré comme une boucle. Par conséquent, les instructions de contrôle de boucles `next`, `last`, ou `redo` ne peuvent être utilisées pour quitter ou refaire le bloc.

exec LISTE

exec PROGRAMME LISTE

La fonction `exec()` exécute une commande système *et ne retourne jamais* — utilisez `system()` à la place de `exec()` si vous souhaitez qu'elle retourne. Elle échoue et retourne FAUX si et seulement si la commande n'existe pas *et* est exécutée directement plutôt que par votre interpréteur de commandes (shell) (cf. ci-dessous).

Comme c'est une erreur courante d'utiliser `exec()` au lieu de `system()`, Perl vous alerte si l'expression suivante n'est pas `die()`, `warn()`, ou `exit()` (si `-w` est utilisé — ce que vous faites toujours, évidemment.) Si vous voulez *vraiment* faire suivre le `exec()` d'une autre expression, vous pouvez utiliser l'une de ces méthodes pour éviter l'avertissement :

```
exec ('foo') or print STDERR "couldn't exec foo: $!";
{ exec ('foo') }; print STDERR "couldn't exec foo: $!";
```

S'il y a plus d'un argument dans la LISTE, ou si la LISTE est un tableau de plus d'une valeur, appelez `execvp(3)` avec les arguments de la LISTE. S'il n'y a qu'un seul argument scalaire ou un tableau à un seul élément et que cet argument contient des méta-caractères du shell, l'argument entier est passé à l'interpréteur de commandes shell pour son expansion et son exécution (il s'agit de `/bin/sh -c` sur les plates-formes Unix, mais cela varie en fonction des plates-formes.) S'il n'y a aucun méta-caractères du shell dans l'argument, il est découpé en mots et passé directement à `execvp()` qui est plus efficace.

Exemples :

```
exec '/bin/echo', 'Your arguments are: ', @ARGV;
exec "sort $outfile | uniq";
```

Si vous ne souhaitez pas vraiment exécuter le premier argument mais plutôt l'utiliser pour dissimuler le nom du programme réellement utilisé, vous pouvez spécifier le vrai programme à exécuter comme un ``objet indirect'' (sans virgule) au début de la LISTE. (Ceci force toujours l'interprétation de la LISTE comme une liste multivaluée, même s'il n'y a qu'un seul scalaire dedans.) Exemple :

```
$shell = '/bin/csh';
exec $shell '-sh';           # prétend qu'il s'agit d'un login shell
```

ou, plus directement,

```
exec { '/bin/csh' } '-sh';   # prétend qu'il s'agit d'un login shell
```

Quand les arguments sont exécutés par le shell système, les résultats seront dépendants de ses caprices et de ses capacités. Voir CHAÎNE` in the perlop manpage pour plus de détails. Utiliser un objet indirect avec exec() ou system() est aussi plus sûr. Cet usage force l'interprétation des arguments comme une liste multivaluée, même si elle ne contient qu'un seul élément. De cette façon, vous ne risquez rien des jokers du shell ou de la séparation des mots par des espaces.

```
@args = ( "echo surprise" );
system @args;           # sujet à des échappement du shell
                        # si @args == 1
system { $args[0] } @args; # sûr même avec une liste à un seul élément
```

La première version, celle sans l'objet indirect, exécute le programme *echo*, en lui passant "surprise" comme argument. La seconde version ne le fait pas -- elle essaie d'exécuter un programme littéralement appelé *echo surprise*, ne le trouve pas et assigne à \$? une valeur non nulle indiquant une erreur.

Depuis la version v5.6.0, Perl essaie avant d'effectuer le exec() de vider tous les tampons des fichiers ouverts en écriture mais ce n'est pas le cas sur toutes les plates-formes (voir the perlport manpage). Pour être plus sûr, vous devriez positionner la variable \$| (\$AUTOFLUSH en anglais) ou appeler la méthode autoflush() des objets IO: :Handle pour chacun des descripteurs ouverts afin d'éviter toute perte de données.

Remarquez que exec() n'exécutera ni vos blocs END ni les méthodes DESTROY de vos objets.

exists EXPR

Si EXPR spécifie un élément d'une table de hachage ou d'un tableau, retourne VRAI si cet élément existe, même si la valeur correspondante est indéfinie. L'élément n'est pas créé automatiquement s'il n'existe pas.

```
print "Exists\n"      if exists $array{$key};
print "Defined\n"    if defined $array{$key};
print "True\n"       if $array{$key};
print "Exists\n"     if exists $hash{$key};
print "Defined\n"    if defined $hash{$key};
print "True\n"       if $hash{$key};
print "Exists\n"     if exists $array[$index];
print "Defined\n"    if defined $array[$index];
print "True\n"       if $array[$index];
```

Un élément d'une table de hachage ou d'un tableau ne peut être VRAI que s'il est défini, et défini que s'il existe, mais la réciproque n'est pas nécessairement vraie.

Si EXPR spécifie le nom d'une sous-routine, retourne VRAI si la sous-routine spécifiée a déjà été déclarée, même si elle est toujours indéfinie. Utiliser un nom de sous-routine pour tester si elle existe (exists) ou si elle est définie (defined) ne compte pas comme une déclaration.

```
print "Exists\n"      if exists &subroutine;
print "Defined\n"    if defined &subroutine;
```

Notez que l'expression EXPR peut être arbitrairement compliquée tant qu'au final, elle désigne une sous-routine ou un élément d'une table de hachage ou d'un tableau :

```
if (exists $ref->{A}->{B}->{$key}) { }
```


perlfunc – Fonctions Perl prédéfinies

```
if (exists $hash{A}{B}{$key})      { }
if (exists $ref->{A}->{B}->[$ix])  { }
if (exists $hash{A}{B}[$ix])      { }
if (exists &{$ref->{A}{B}{$key}})  { }
```

Bien que l'élément le plus profond ne soit pas soudainement créé juste parce son existence a été testée, les éléments intermédiaires, eux, le seront. Par conséquent, `$ref->{"A"}` et `$ref->{"A"}->{"B"}` seront créés à cause du test d'existence de l'élément lié à la clé `$key` ci-dessus. Cela arrivera à chaque fois que l'opérateur flèche est utilisé, y compris dans le cas suivant :

```
undef $ref;
if (exists $ref->{"Some key"})      { }
print $ref;                        # affiche HASH(0x80d3d5c)
```

Cette génération spontanée un peu surprenante qui, au premier coup d'oeil (ni même au second d'ailleurs), n'est pas dans le contexte d'une lvalue pourrait être supprimée dans une version future.

Voir [Pseudo-tables de hachage : utiliser un tableau comme table de hachage in the perlref manpage](#) pour des informations spécifiques concernant l'utilisation de `exists()` sur les pseudo-hachages.

L'utilisation d'un appel à une subroutine à la place du nom de cette subroutine comme argument de `exists()` est une erreur.

```
exists &sub;                        # OK
exists &sub();                      # Erreur
```

exit *EXPR*

Évalue *EXPR* puis quitte immédiatement avec cette valeur. Exemple :

```
$ans = <STDIN>;
exit 0 if $ans =~ /^[Xx]/;
```

Voir aussi `die()`. Si l'expression *EXPR* est omise, quitte avec le statut 0. Les seules valeurs universellement reconnues pour *EXPR* sont 0 en cas de réussite et 1 en cas d'erreur ; toutes les autres valeurs sont sujettes à des interprétations imprévisibles, en fonction de l'environnement dans lequel le programme Perl est exécuté. Par exemple, terminer un filtre de messages entrants de *sendmail* avec comme valeur 69 (EX_UNAVAILABLE) provoquera la non livraison du message, mais ce n'est pas vrai partout.

Vous ne devriez pas utiliser `exit()` pour interrompre une routine s'il existe une chance pour que quelqu'un souhaite intercepter une erreur qui arrive. Utilisez `die()` à la place, qui peut être intercepté par un `eval()`.

La fonction `exit()` ne termine pas toujours le process immédiatement. Elle appelle d'abord toutes les routines `END` définies, mais ces routines `END` ne peuvent pas annuler la terminaison. De la même façon, tout destructeur d'objet qui doit être appelé le sera avant la sortie. Si cela pose problème, vous pouvez appelé `POSIX::_exit($status)` pour éviter le traitement des destructeurs et des subroutine `END`. Voir [the perlmod manpage](#) pour les détails.

exp *EXPR*

exp

Retourne *e* (la base des logarithmes naturels ou népérien) élevé à la puissance *EXPR*. Si *EXPR* est omis, retourne `exp($_)`.

fcntl *DESCRIPTEUR, FONCTION, SCALAIRE*

Implémente la fonction `fcntl(2)`. Vous devrez probablement d'abord écrire

```
use Fcntl;
```

pour avoir les définitions de constantes correctes. Le traitement de l'argument et la valeur de retour se fait exactement de la même manière que `ioctl()` plus bas. Par exemple :

```
use Fcntl;
fcntl($filehandle, F_GETFL, $packed_return_buffer)
    or die "can't fcntl F_GETFL: $!";
```

Vous n'avez pas à vérifier le résultat de `fcntl()` via `defined()`. Comme `ioctl()`, elle transforme le retour 0 de l'appel système en un `0 but true` en Perl. Cette chaîne est vraie dans un contexte booléen et vaut 0 dans un contexte numérique. Elle est aussi exempte des alertes habituelles de `-w` sur les conversions numériques impropres.

Notez que `fcntl()` produira une erreur fatale si elle est utilisée sur une machine n'implémentant pas `fcntl(2)`. Voir le module `Fcntl` ou `fcntl(2)` pour connaître les fonctions disponibles sur votre système.

***fileno* DESCRIPTEUR**

Retourne le numéro système associé à un descripteur de fichier. Ceci est utile pour construire des vecteurs pour `select()` et pour les opérations POSIX de manipulation bas niveau de terminaux tty. Si `DESCRIPTEUR` est une expression, la valeur est prise comme un descripteur indirect, généralement son nom.

Vous pouvez utiliser ceci pour déterminer si deux descripteurs se réfèrent au même numéro sous-jacent :

```
if (fileno(THIS) == fileno(THAT)) {
    print "THIS and THAT are dups\n";
}
```

(Les descripteurs connectés à des objets en mémoire via les nouvelles fonctionnalités de `open` peuvent retourner une valeur indéfinie bien qu'ils soient ouverts.)

***flock* DESCRIPTEUR, OPERATION**

Appelle `flock(2)`, ou son émulation, sur le `DESCRIPTEUR`. Retourne `VRAI` en cas de succès, `FAUX` en cas d'échec. Produit une erreur fatale lorsqu'il est utilisé sur une machine n'implémentant pas `flock(2)`, le verrouillage `fcntl(2)`, ou `lockf(3)`. `flock()` est une interface Perl portable de verrouillage de fichiers, bien qu'il ne verrouille que des fichiers en entier, et non pas des enregistrements.

La sémantique non évidente mais néanmoins traditionnelle de `flock` consiste à attendre indéfiniment jusqu'à la libération du verrou. Ce verrou est **purement consultatif**. De tels verrous sont plus faciles d'utilisation mais offrent moins de garantie. Cela signifie que des fichiers verrouillés via `flock` peuvent être modifiés par des programmes n'utilisant pas `flock`. Voir [the perlport manpage](#) ou les pages de manuel de votre système pour la documentation plus détaillée. Il vaut mieux faire avec ce comportement traditionnel si vous visez la portabilité. (Sinon, libre à vous d'utiliser les fonctionnalités ("features") de votre système d'exploitation. Les contraintes de portabilité ne doivent pas vous empêcher de faire ce que voulez.)

`OPERATION` peut être `LOCK_SH`, `LOCK_EX` ou `LOCK_UN`, éventuellement combinée avec `LOCK_NB`. Ces constantes ont traditionnellement pour valeurs 1, 2, 8 et 4, mais vous pouvez utiliser les noms symboliques s'ils sont importés du module `Fcntl`, soit individuellement, soit en tant que groupe en utilisant la balise `:flock`. `LOCK_SH` demande un verrou partagé, `LOCK_EX` demande un verrou exclusif et `LOCK_UN` libère un verrou précédemment demandé. Si `LOCK_NB` est ajouté à `LOCK_SH` ou `LOCK_EX` (via un 'ou' bit à bit) alors `flock()` retournera immédiatement plutôt que d'attendre la libération du verrou (vérifiez le code de retour pour savoir si vous l'avez obtenu).

Pour éviter une mauvaise synchronisation, Perl vide le tampon du `DESCRIPTEUR` avant de le (dé)verrouiller.

Notez que l'émulation construite avec `lockf(3)` ne fournit pas de verrous partagés et qu'il exige que DESCRIPTEUR soit ouvert en écriture. Ce sont les sémantiques qu'implémente `lockf(3)`. La plupart des systèmes (si ce n'est tous) implémentent toutefois `lockf(3)` en termes de verrous `fcntl(2)`, la différence de sémantiques ne devrait donc pas gêner trop de monde.

Notez que l'émulation de `flock(3)` via `fcntl(2)` implique que DESCRIPTEUR soit ouvert en lecture pour l'utilisation de `LOCK_SH` et en écriture pour l'utilisation de `LOCK_EX`.

Notez aussi que certaines versions de `flock()` ne peuvent pas verrouiller des choses à travers le réseau, vous devriez utiliser des appels à `fcntl()` plus spécifiques au système dans ce but. Si vous le souhaitez, vous pouvez contraindre Perl à ignorer la fonction `flock(2)` de votre système et lui fournir ainsi sa propre émulation basée sur `fcntl(2)`, en passant le flag `-Ud_flock` au programme *Configure* lors de la configuration de perl.

Voici un empilage de mails pour les systèmes BSD.

```
use Fcntl ':flock'; # import des constantes LOCK_*
sub lock {
    flock(MBOX, LOCK_EX);
    # et, si quelqu'un ajoute
    # pendant notre attente...
    seek(MBOX, 0, 2);
}
sub unlock {
    flock(MBOX, LOCK_UN);
}
open(MBOX, ">>/usr/spool/mail/$ENV{'USER'}")
    or die "Can't open mailbox: $!";
lock();
print MBOX $msg, "\n\n";
unlock();
```

Sur les systèmes qui possèdent un vrai `flock()`, les verrous sont hérités à travers les appels à `fork()` alors que ceux qui s'appuient sur la fonction `fcntl()` plus capricieuse perdent les verrous, rendant ainsi l'écriture de serveur plus difficile.

Voir aussi *DB_File* pour d'autres exemples avec `flock()`.

fork

Effectue un appel système `fork(2)` pour créer un nouveau processus exécutant le même programme au même point. Retourne l'identifiant (pid) de l'enfant au processus père, 0 au processus fils ou `undef` en cas d'échec. Les descripteurs de fichiers (et parfois les verrous posés sur ces descripteurs) sont partagés sinon tout le reste est recopié. Sur la plupart des systèmes qui supportent l'appel système `fork()`, une grande attention a été portée pour qu'il soit extrêmement efficace (par exemple en utilisant la méthode de copie-à-l'écriture sur les pages mémoires contenant des données). C'est donc la paradigme dominant pour la gestion du multi-tâches durant les dernières décennies.

Depuis la version v5.6.0, Perl tente de vider les tampons de tous les fichiers ouverts en écriture avant d'effectuer le `fork()` mais cela n'est pas supporté sur toutes les plates-formes (voir [the perlport manpage](#)). Pour être plus sûr, vous devriez positionner la variable `$|` (`$AUTOFLUSH` en anglais) ou appeler la méthode `autoflush()` des objets `IO::Handle` pour chacun des descripteurs ouverts afin d'éviter toute duplication de données.

Si vous dupliquez avec `fork()` sans attendre votre fils, vous allez accumuler des zombies. Sur certains systèmes, vous pouvez éviter cela en positionnant `$$SIG{CHLD}` à "IGNORE". Voir aussi [the perlipc manpage](#) pour des exemples d'utilisation de `fork()` et de suppression d'enfants moribonds.

Notez que si votre fils dupliqué hérite de descripteurs de fichier systèmes, tels que STDIN et STDOUT, qui sont en fait connectés par un tube ou une socket, même si vous sortez du programme, alors le serveur distant (disons tels que httpd ou rsh) ne saura pas que vous avez terminé. Vous devriez les réouvrir vers `/dev/null` en cas de problème.

format

Déclare un format visuel utilisable par la fonction [write\(\)](#). Par exemple :

```
format Something =
    Test: @<<<<<<<< @|||  @>>>>>>
           $str,      $%,      '$' . int($num)
.
$str = "widget";
$num = $cost/$quantity;
$~ = 'Something';
write;
```

Voir [the perform manpage](#) pour de nombreux détails et exemples.

formline IMAGE,LISTE

C'est une fonction interne utilisée par les formats ([format](#)), bien que vous puissiez aussi l'appeler. Elle formate (voir [the perform manpage](#)) une liste de valeurs selon le contenu de IMAGE, plaçant la sortie dans l'accumulateur du format de sortie `^A` (ou `$ACCUMULATOR` en anglais). Finalement, lorsqu'un [write\(\)](#) est effectué, le contenu de `^A` est écrit dans un descripteur de fichier, mais vous pouvez aussi lire `^A` vous-même et réassigner " " à `^A`. Notez qu'un format typique appelle [formline](#) une fois par ligne du formulaire mais la fonction [formline](#) elle-même ne se préoccupe pas du nombre de passage à la ligne inclus dans l'IMAGE. Ceci signifie que `~` et `~~` traiteront l'IMAGE complète comme une seule ligne. Vous pouvez donc utiliser de multiples [formline](#) pour implémenter un seul format d'enregistrement, tout comme le compilateur de format.

Faites attention si vous utilisez des guillemets autour de l'image, car un caractère ``@` pourrait être pris pour le début d'un nom de tableau. [formline\(\)](#) retourne toujours VRAI. Cf. [the perform manpage](#) pour d'autres exemples.

getc DESCRIPTEUR

getc

Retourne le prochain caractère du fichier d'entrée attaché au DESCRIPTEUR ou la valeur indéfinie (undef) à la fin du fichier ou en cas d'échec. Si le DESCRIPTEUR est omis, utilise STDIN. Ce n'est pas particulièrement efficace. De plus, ce n'est pas utilisable tel quel pour obtenir des caractères sans que l'utilisateur presse ENTER. Pour ça, essayez quelque chose comme :

```
if ($BSD_STYLE) {
    system "stty cbreak </dev/tty >/dev/tty 2>&1";
}
else {
    system "stty", '-icanon', 'eol', "\001";
}
$key = getc(STDIN);
if ($BSD_STYLE) {
    system "stty -cbreak </dev/tty >/dev/tty 2>&1";
}
else {
    system "stty", 'icanon', 'eol', '^@'; # ASCII null
}
print "\n";
```

Déterminer la valeur de `$BSD_STYLE` est laissé en exercice au lecteur.

La fonction `POSIX::getattr()` peut faire ceci de façon plus portable sur des systèmes compatibles POSIX. Voir aussi le module `Term::ReadKey` de votre site CPAN le plus proche. Les détails sur CPAN peuvent être trouvés dans [CPAN in the perlmodlib manpage](#).

getlogin

Implémente la fonction de la bibliothèque C portant le même nom et qui, sur la plupart des systèmes, retourne le login courant à partir de `/etc/utmp`, si il existe. Si l'appel retourne null, utilisez [getpwuid\(\)](#).

```
$login = getlogin || getpwuid($<) || "Kilroy";
```

N'utilisez pas [getlogin\(\)](#) pour de l'authentification : ce n'est pas aussi sûr que [getpwuid\(\)](#).

getpeername SOCKET

Renvoie l'adresse `sockaddr` compactée (voir [pack\(\)](#)) de l'autre extrémité de la connexion SOCKET.

```
use Socket;
$hersockaddr = getpeername(SOCK);
($port, $iaddr) = unpack_sockaddr_in($hersockaddr);
$herhostname = gethostbyaddr($iaddr, AF_INET);
$herstraddr = inet_ntoa($iaddr);
```

getpgrp PID

Renvoie le groupe courant du processus dont on fournit le PID. Utilisez le PID 0 pour obtenir le groupe courant du processus courant. Cela engendra une exception si on l'utilise sur une machine qui n'implémente pas `getpgrp(2)`. Si PID est omis, renvoie le groupe du processus courant. Remarquez que la version POSIX de [getpgrp\(\)](#) ne prend pas d'argument PID donc seul `PID==0` est réellement portable.

getppid

Renvoie l'id du processus parent.

getpriority WHICH,WHO

Renvoie la priorité courant d'un processus, d'un groupe ou d'un utilisateur. (Voir [getpriority\(2\)](#).) Cela engendra une exception si on l'utilise sur une machine qui n'implémente pas `getpriority(2)`.

getpwnam NAME

getgrnam NAME

gethostbyname NAME

getnetbyname NAME

getprotobyname NAME

getpwuid UID

getgrgid GID

getservbyname NAME,PROTO

gethostbyaddr ADDR,ADDRTYPE

getnetbyaddr ADDR,ADDRTYPE

getprotobynumber NUMBER

getservbyport PORT,PROTO

getpwent

getgrent

gethostent

getnetent

getprotoent
getservent
setpwent
setgrent
sethostent STAYOPEN
setnetent STAYOPEN
setprotoent STAYOPEN
setservent STAYOPEN
endpwent
endgrent
endhostent
endnetent
endprotoent
endservent

Ces routines réalisent exactement les mêmes fonctions que leurs homologues de la bibliothèque système. Dans un contexte de liste, les valeurs retournées par les différentes routines sont les suivantes :

```

($name, $passwd, $uid, $gid,
 $quota, $comment, $gcos, $dir, $shell, $expire) = getpw*
($name, $passwd, $gid, $members) = getgr*
($name, $aliases, $addrtype, $length, @addrs) = gethost*
($name, $aliases, $addrtype, $net) = getnet*
($name, $aliases, $proto) = getproto*
($name, $aliases, $port, $proto) = getserv*

```

(Si une entrée n'existe pas, vous récupérerez une liste vide.)

La signification exacte du champ `$gcos` varie mais contient habituellement le nom réel de l'utilisateur (au contraire du nom de login) et d'autres informations pertinentes pour cet utilisateur. Par contre, sachez que sur de nombreux systèmes, les utilisateurs peuvent changer eux-mêmes ces informations. Ce n'est donc pas une information de confiance. Par conséquent `$gcos` est souillée (voir [the perlsec manpage](#)). Les champs `$passwd`, `$shell` ainsi que l'interpréteur de commandes (login shell) et le mot de passe crypté sont aussi souillés pour les mêmes raisons.

Dans un contexte scalaire, vous obtenez le nom sauf lorsque la fonction fait une recherche par nom auquel cas vous récupérerez autre chose. (Si une entrée n'existe pas vous récupérerez la valeur undef.) Par exemple :

```

$uid    = getpwnam($name);
$name   = getpwuid($num);
$name   = getpwent();
$gid    = getgrnam($name);
$name   = getgrgid($num);
$name   = getgrent();
#etc.

```

Dans `getpw*`(*)*, les champs `$quota`, `$comment` et `$expire` sont des cas spéciaux dans le sens où ils ne sont pas supportés sur de nombreux systèmes. Si `$quota` n'est pas supporté, c'est un scalaire vide. Si il est supporté, c'est habituellement le quota disque. Si le champ `$comment` n'est pas supporté, c'est un scalaire vide. Si il est supporté, c'est habituellement le commentaire « administratif » associé à l'utilisateur. Sur certains systèmes, le champ `$quota` peut être `$change` ou `$age`, des champs qui sont en rapport avec le vieillissement du mot de passe. Sur certains systèmes, le champ `$comment` peut être `$class`. Le champ `$expire`, s'il est présent, exprime la période d'expiration du compte ou du mot de passe. Pour connaître la disponibilité et le sens exact de tous ces champs sur votre système, consultez votre documentation de `getpwnam(3)` et le fichier `pwd.h`. À partir de Perl, vous pouvez trouver le sens de vos champs `$quota` et `$comment`

et savoir si vous avez le champ `$expire` en utilisant le module `Config` pour lire les valeurs de `d_pwquota`, `d_pwage`, `d_pwchange`, `d_pwcomment` et `d_pwexpire`. Les fichiers de mots de passe cachés (shadow password) ne sont supportés que si l'implémentation de votre système est faite telle que les appels aux fonctions normales de la bibliothèque C accèdent à ces fichiers lorsque vos privilèges vous y autorisent. Toute autre implémentation incorrecte via des appels à une bibliothèque séparée n'est pas supportée.

La valeur `$members` renvoyée par les fonctions `getgr*`() est la liste des noms de login des membres du groupe séparés par des espaces.

Pour les fonctions `gethost*`(), si la variable `h_errno` est supportée en C, sa valeur sera retournée via `$?` si l'appel à la fonction échoue. La valeur de `@addr` qui est retournée en cas de succès est une liste d'adresses à plat telle que retournée par l'appel système correspondant. Dans le domaine Internet, chaque adresse fait quatre octets de long et vous pouvez la décompacter en disant quelque chose comme :

```
($a,$b,$c,$d) = unpack('C4',$addr[0]);
```

Si vous êtes fatigué de devoir vous souvenir que tel élément de la liste retournée correspond à telle valeur, des interfaces par nom sont fournies par les modules `File::stat`, `Net::hostent`, `Net::netent`, `Net::protoent`, `Net::servent`, `Time::gmtime`, `Time::localtime` et `User::grent`. Ils remplacent les appels internes normaux par des versions qui renvoient des objets ayant le nom approprié pour chaque champ. Par exemple :

```
use File::stat;
use User::pwent;
$sis_his = (stat($filename)->uid == pwent($whoever)->uid);
```

Bien que les deux appels se ressemblent (appel à la méthode 'uid'), ce n'est pas la même chose car l'objet `File::stat` est différent de l'objet `User::pwent`.

getsockname SOCKET

Renvoie l'adresse `sockaddr` compactée de cette extrémité de la connexion `SOCKET` si vous ne connaissez pas cette adresse car vous avez différentes adresses IP utilisables pour établir cette connexion.

```
use Socket;
$mysockaddr = getsockname(SOCK);
($port, $myaddr) = sockaddr_in($mysockaddr);
printf "Connect to %s [%s]\n",
       scalar gethostbyaddr($myaddr, AF_INET),
       inet_ntoa($myaddr);
```

getsockopt SOCKET,LEVEL,OPTNAME

Renvoie l'option demandée de `SOCKET` ou `undef` en cas d'erreur.

glob EXPR

glob

Dans un contexte de liste, retourne la liste (éventuellement vide) des fichiers correspondants à l'expansion de la valeur de `EXPR` telle que le shell standard Unix `/bin/csh` la ferait. Dans un contexte scalaire, `glob` produit les éléments de cette liste un par un jusqu'à l'épuiser puis il retourne `undef`. C'est la fonction interne qui implémente l'opérateur `<* . c>` mais vous pouvez l'utiliser directement. Si `EXPR` est omis, `$_` est utilisé à la place. L'opérateur `<* . c>` est présenté plus en détail dans [Les opérateurs d'E/S in the perlop manpage](#).

Depuis la version v5.6.0, cet opérateur est implémenté via l'extension standard `File::Glob`. Voir [the File::Glob manpage](#) pour plus de détails.

gmtime EXPR

perlfunc – Fonctions Perl prédéfinies

Convertit une date telle que celle retournée par la fonction `time` en un tableau de 9 éléments avec la date liée au fuseau horaire du méridien de Greenwich standard. On l'utilise typiquement de la manière suivante :

```
# 0 1 2 3 4 5 6 7 8
($sec,$min,$heure,$mjour,$mois,$annee,$sjour,$ajour,$isdst) =
    gmtime(time);
```

Tous les éléments du tableau sont numériques et restent tels qu'ils apparaissent dans la structure 'struct tm'. `$sec`, `$min` et `$heure` sont les secondes, les minutes et l'heure de l'instant spécifié. `$mjour` est le quantième du mois et `$mois` est le mois lui-même, dans l'intervalle 0..11 avec 0 pour janvier et 11 pour décembre. `$annee` est le nombre d'années depuis 1900 et donc `$annee` vaut 123 en l'an 2023. `$sjour` est le jour de la semaine avec 0 pour le dimanche et 3 pour le mercredi. `$ajour` est le jour de l'année dans l'intervalle 1..365 (ou 1..366 pour les années bissextiles.) `$isdst` est vrai si l'heure d'été est en cours à la date spécifiée et faux sinon.

Remarquez bien que `$annee` n'est pas que les deux derniers chiffres de l'année. Si vous supposez cela, vous créez des programmes non compatible an 2000 -- et vous ne voulez pas faire cela, n'est-ce pas ?

La bonne méthode pour obtenir une année complète sur 4 chiffres est tout simplement :

```
$annee += 1900;
```

Et pour obtenir les deux derniers chiffres de l'année (e.g. '01' en 2001):

```
$annee = sprintf("%02d", $annee % 100);
```

Si `EXPR` est omis, calcule `gmtime(time)`.

Dans un contexte scalaire, retourne la valeur de `ctime(3)` :

```
$now_string = gmtime; # e.g., "Thu Oct 13 04:54:34 1994"
```

Voir aussi la fonction `timegm()` fournie par le module `Time::Local` et la fonction `strftime(3)` disponible via le module POSIX.

La valeur scalaire n'est **pas** dépendante du locale, voir [the perllocale manpage](#), mais est construite en interne par Perl. Voir aussi le module `Time::Local` et les fonctions `strftime(3)` et `mktime(3)` du module POSIX. Pour obtenir quelque chose de similaire mais dont les chaînes de caractères des dates dépendent du locale, paramétrez vos variables d'environnement liées au locale (voir [the perllocale manpage](#)) et essayez par exemple :

```
use POSIX qw(strftime);
$now_string = strftime "%a %b %e %H:%M:%S %Y", gmtime;
```

Notez que les séquences `%a` et `%b` qui sont les formes courtes du jour de la semaine et du mois de l'année, n'ont pas obligatoirement 3 caractères de long dans tous les locale.

goto LABEL

goto EXPR

goto &NAME

La forme `goto LABEL` trouve la ligne étiquetée par `LABEL` et continue l'exécution à cette ligne. On ne peut pas l'utiliser pour sauter à l'intérieur d'une construction qui nécessite une initialisation telle qu'une subroutine ou une boucle `foreach`. On ne peut pas non plus l'utiliser pour aller dans une construction qui peut être optimisée ou pour sortir d'un bloc ou d'une subroutine donné à `sort()`. On peut par contre l'utiliser pour aller n'importe où dans la portée actuelle, même pour sortir des subroutines, bien qu'il soit meilleur d'utiliser pour cela d'autres constructions telles que `last` ou `die()`. L'auteur de Perl n'a jamais ressenti le besoin d'utiliser cette forme de `goto` (en Perl bien sûr -- en C, c'est une autre histoire). (La différence avec le C provient du fait que le C n'offre pas le nommage des boucles couplée aux instructions de contrôle de boucles. Cette possibilité de Perl remplace la quasi totalité des usages structurés du `goto` dans d'autres langages.)

La forme `goto-EXPR` attend un nom d'étiquette dont la portée sera résolue dynamiquement. Cela autorise les `goto` calculés à la FORTRAN mais ce n'est pas vraiment recommandé si vous vous souciez de la maintenance.

```
goto ("FOO", "BAR", "GLARCH")[$i];
```

La forme `goto-&NAME` est vraiment différente des autres formes de `goto`. En fait, ce n'est pas du tout un `goto` et il n'a donc pas les stigmates habituellement associés aux autres `goto`. Cette forme substitue l'appel de la sous-routine en cours d'exécution (en oubliant toutes les modifications faites par des appels à `local()`) par un appel à la sous-routine donnée en argument. C'est utilisé par les routines `AUTOLOAD` qui désirent charger une autre sous-routine puis tout faire comme si c'était cette autre sous-routine qui avait réellement été appelée (sauf que toutes les modifications faites à `@_` dans la sous-routine courante sont propagées à l'autre sous-routine). Après le `goto`, pas même `caller()` n'est capable de s'apercevoir que la sous-routine initiale a été appelée au préalable.

`NAME` n'est pas nécessairement le nom d'une sous-routine ; cela peut être une variable scalaire contenant une référence à du code ou un bloc dont l'évaluation produit une référence à du code.

grep BLOC LISTE

grep EXPR,LISTE

Cette fonction est similaire dans l'esprit mais pas identique à `grep(1)` et tous ses dérivés. En particulier, elle n'est pas limitée à l'utilisation d'expressions rationnelles.

Elle évalue le bloc `BLOC` ou l'expression `EXPR` pour chaque élément de `LISTE` (qui est localement lié à `$_`) et retourne la liste des valeurs constituée des éléments pour lesquels l'expression est évaluée à `true` (vrai). Dans un contexte scalaire, retourne le nombre de fois où l'expression est vraie (`true`).

```
@foo = grep(!/^#/ , @bar); # supprime les commentaires
```

ou de manière équivalente :

```
@foo = grep {!/^#/} @bar; # supprime les commentaires
```

Remarquez que, puisque `$_` est un référence dans la liste de valeurs, il peut être utilisé pour modifier les éléments du tableau. Bien que ce soit supporté et parfois pratique, cela peut aboutir à des résultats bizarres si `LISTE` n'est pas un tableau nommé. De manière similaire, `grep` renvoie des alias de la liste originale exactement comme le fait une variable de boucle `for`. Et donc, modifier un élément d'une liste retournée par `grep` (par exemple dans un `foreach`, un `map()` ou un autre `grep()`) modifie réellement l'élément de la liste originale. Voir aussi `map` pour obtenir un tableau composé des résultats de `BLOC` ou `EXPR`.

hex EXPR

hex

Interprète `EXPR` comme une chaîne hexadécimale et retourne la valeur correspondante. (Pour convertir des chaînes qui commencent soit par 0 soit par 0x voir `oct()`.) Si `EXPR` est omis, c'est `$_` qui est utilisé.

```
print hex '0xAf'; # affiche '175'
print hex 'aF'; # idem
```

Les chaînes hexadécimales ne peuvent représenter que des entiers. Les chaînes qui provoquent un dépassement de la capacité des entiers déclenchent un avertissement. Au contraire de `oct()`, les espaces initiaux ne sont pas ignorés.

import

Il n'existe pas de fonction interne `import()`. C'est juste une méthode ordinaire (une sous-routine) définie (ou héritée) par les modules qui veulent exporter des noms vers d'autres modules. La fonction `use()` appelle la méthode `import()` du paquetage utilisé. Voir aussi `use()`, `the perlmod manpage` et `Exporter`.

index CHAINE,SUBSTR,POSITION

index CHAINE,SUBSTR

La fonction `index()` recherche une chaîne dans une autre mais sans les fonctionnalités génériques de reconnaissance de motifs des expressions rationnelles. Retourne la position de la première occurrence du SUBSTR dans CHAINE à partir de POSITION inclus. Si POSITION est omis, le recherche commence au début de la chaîne. La valeur retournée est relative à 0 (ou à la valeur que vous avez affectée à la variable `$[--` mais ce n'est pas à faire). Si la sous-chaîne SUBSTR n'est pas trouvée, la valeur retournée est la valeur de référence moins 1 (habituellement -1).

int EXPR

int

Retourne la partie entière de EXPR. Si EXPR est omis, c'est `$_` qui est utilisé. Vous ne devriez pas utiliser cette fonction pour faire des arrondis parce qu'elle tronque la valeur vers 0 et parce que la représentation interne des nombre en virgule flottante produit parfois des résultats contre-intuitifs. Par exemple `int(-6.725/0.025)` produit -268 au lieu de -269 ; C'est dû aux erreurs de calcul qui donne un résultat comme -268.99999999999994315658. Habituellement `sprintf()` et `printf()` ou les fonctions POSIX: `floor` et `ceil` vous seront plus utiles que `int()`.

ioctl DESCRIPTEUR,FONCTION,SCALAIRE

Implémente la fonction `ioctl(2)`. Vous aurez probablement à dire :

```
require "ioctl.ph"; # probablement dans /usr/local/lib/perl/ioctl.ph
```

en premier lieu pour obtenir les définitions correctes des fonctions. Si `ioctl.ph` n'existe pas ou ne donne pas les définitions correctes, vous devrez les fournir vous-même en vous basant sur les fichiers d'en-tête C tels que `<sys/ioctl.h>`. (Il existe un script Perl appelé **h2ph** qui vient avec le kit Perl et qui devrait vous aider à faire cela mais il n'est pas trivial.) SCALAIRE sera lu et/ou modifié selon la fonction FONCTION -- un pointeur sur la valeur alphanumérique de SCALAIRE est passé comme troisième argument du vrai appel système `ioctl()`. (Si SCALAIRE n'a pas de valeur alphanumérique mais a une valeur numérique, c'est cette valeur qui sera passée plutôt que le pointeur sur la valeur alphanumérique. Pour garantir que c'est bien ce qui se passera, ajouter 0 au scalaire avant de l'utiliser.) Les fonction `pack()` et `unpack()` permettent de manipuler les différentes structures utilisées par `ioctl()`.

Les différentes valeurs retournées par `ioctl` (et `fcntl`) sont les suivantes :

si l'OS retourne :	alors Perl retournera :
-1	la valeur indéfinie (undef)
0	la chaîne "0 but true"
autre nombre	ce nombre

Ainsi Perl retourne vrai en cas de succès et faux en cas d'échec et vous pouvez encore déterminer la véritable valeur retourné par le système d'exploitation :

```
$retval = ioctl(...) || -1;
printf "System returned %d\n", $retval;
```

La chaîne spéciale `"0 but true"` est traitée par `-w` comme une exception qui ne déclenche pas le message d'avertissement concernant les conversions numériques incorrectes.

Voici un exemple qui rend non bloquant au niveau système le descripteur de fichier REMOTE.

En revanche, vous aurez à gérer vous-même le traitement de `$|`.

```
use Fcntl qw(F_GETFL F_SETFL O_NONBLOCK);
$flags = fcntl(REMOTE, F_GETFL, 0)
    or die "Can't get flags for the socket: $!\n";
$flags = fcntl(REMOTE, F_SETFL, $flags | O_NONBLOCK)
    or die "Can't set flags for the socket: $!\n";
```

join *EXPR,LISTE*

Concatène les différentes chaînes de LISTE en une seule chaîne où les champs sont séparés par la valeur de EXPR et retourne cette nouvelle chaîne. Exemple :

```
$_ = join(':', $login,$passwd,$uid,$gid,$gcos,$home,$shell);
```

Au contraire de [split](#), [join](#) ne prend pas un motif comme premier argument. À comparer à [split](#).

keys *HASH*

Retourne une liste constituée de toutes les clés (en anglais : keys) de la table de hachage fournie. (Dans un contexte scalaire, retourne le nombre de clés.) Les clés sont produites dans un ordre apparemment aléatoire. Cet ordre particulier pourrait changer dans une version future mais il est garanti que ce sera toujours le même que celui produit par les fonctions [values\(\)](#) ou [each\(\)](#) (en supposant que la table de hachage n'a pas été modifiée). Un effet de bord est la réinitialisation de l'itérateur de HASH.

Voici encore une autre manière d'afficher votre environnement :

```
@keys = keys %ENV;
@values = values %ENV;
while (@keys) {
    print pop(@keys), '=', pop(@values), "\n";
}
```

et comment trier tout cela par ordre des clés :

```
foreach $key (sort(keys %ENV)) {
    print $key, '=', $ENV{$key}, "\n";
}
```

Pour trier une table de hachage par valeur, vous aurez à utiliser la fonction [sort\(\)](#). Voici le tri d'une table de hachage par ordre décroissant de ses valeurs :

```
foreach $key (sort { $hash{$b} <=> $hash{$a} } keys %hash) {
    printf "%4d %s\n", $hash{$key}, $key;
}
```

En tant que lvalue (valeur modifiable), [keys\(\)](#) vous permet de d'augmenter le nombre de réceptacles alloués pour la table de hachage concernée. Cela peut améliorer les performances lorsque vous savez à l'avance qu'une table va grossir. (C'est tout à fait similaire à l'augmentation de taille d'un tableau en affectant une grande valeur à \$#tableau.)

Si vous dites :

```
keys %hash = 200;
```

alors %hash aura au moins 200 réceptacles alloués — 256 en fait, puisque la valeur est arrondie à la puissance de deux immédiatement supérieure. Ces réceptacles seront conservés même si vous faites %hash = (). Utilisez undef %hash si vous voulez réellement libérer l'espace alloué. En revanche, vous ne pouvez pas utiliser cette méthode pour réduire le nombre de réceptacles alloués (n'ayez aucune inquiétude si vous le faites tout de même par inadvertance : cela n'a aucun effet).

Voir aussi [each](#), [values](#) et [sort](#).

kill *SIGNAL, LISTE*

Envoie un signal à une liste de processus. Retourne le nombre de processus qui ont été correctement « signalés » (qui n'est pas nécessairement le même que le nombre de processus à qui le signal a été envoyé).

```
$cnt = kill 1, $child1, $child2;
kill 9, @goners;
```

Si SIGNAL vaut zéro, aucun signal n'est envoyé. C'est un moyen pratique de vérifier qu'un process existe encore et n'a pas changé son UID. Voir [the perlport manpage](#) pour vérifier la portabilité d'une telle construction.

Au contraire du shell, en Perl, si *SIGNAL* est négatif, il « kill » le groupe de processus plutôt que les processus. (Sur System V, un numéro de *PROCESSUS* négatif « kill » aussi les groupes de processus mais ce n'est pas portable.) Cela signifie que vous utiliserez habituellement une valeur positive comme signal. Vous pouvez aussi utiliser un nom de signal entre apostrophes. Voir [Signaux in the perlipc manpage](#) pour tous les détails.

last LABEL

last

La commande `last` est comme l'instruction `break` en C (telle qu'elle est utilisée dans les boucles) ; cela permet de sortir immédiatement de la boucle en question. En l'absence de LABEL, la commande se réfère à la boucle englobante la plus profonde. Le bloc `continue`, s'il existe, n'est pas exécuté :

```
LINE: while (<STDIN>) {
    last LINE if /^$/;      # exit when done with header
    #...
}
```

`last` ne peut pas être utilisé pour sortir d'un bloc qui doit retourner une valeur comme `eval {}`, `sub {}` ou `do {}` et ne devrait pas être utilisé pour sortir d'une opération `grep()` ou `map()`.

Notez qu'un bloc en lui-même est sémantiquement équivalent à une boucle qui ne s'exécuterait qu'une seule fois. Par conséquent, `last` peut être utilisé pour sortir prématurément d'un tel bloc.

Voir aussi `continue` pour une illustration du comment marche `last`, `next` et `redo`.

lc EXPR

lc

Retourne une version de EXPR entièrement en minuscules. C'est la fonction interne qui implémente le séquence d'échappement `\L` dans les chaînes entre guillemets. Respecte le locale courant `LC_CTYPE` si `use locale` est actif. Voir [the perllocale manpage](#) et [perlunicode](#) pour des plus amples informations concernant la gestion des ``locale" et d'Unicode.

En l'absence de EXPR, s'applique à `$_`.

lcfirst EXPR

lcfirst

Retourne un version de EXPR avec le premier caractère en minuscule. C'est la fonction interne qui implémente le séquence d'échappement `\l` dans les chaînes entre guillemets. Respecte le locale courant `LC_CTYPE` si `use locale` est actif. Voir [the perllocale manpage](#) et [perlunicode](#) pour des plus amples informations concernant la gestion des ``locale" et d'Unicode.

En l'absence de EXPR, s'applique à `$_`.

length EXPR

length

Retourne la longueur en octets de la valeur de EXPR. En l'absence de EXPR, s'applique à `$_`. Notez que cette fonction ne s'applique ni à un tableau ni à une table de hachage pour savoir combien d'éléments ils contiennent. Pour cela, utilisez respectivement `scalar @tableau` et `scalar keys %hash`

link OLDFILE,NEWFILE

Créer un nouveau fichier NEWFILE lié à l'ancien fichier OLDFILE. Retourne true (vrai) en

cas de succès ou false (faux) sinon.

listen SOCKET,QUEUESIZE

Fait exactement la même chose que l'appel système du même nom. Retourne true (vrai) en cas de succès ou false (faux) sinon. Voir les exemples dans [Sockets : Communication Client/Serveur in the perlipc manpage](#).

local EXPR

Vous devriez certainement utiliser `my()` à la place car `local()` n'a pas la sémantique que la plupart des gens accorde à la notion « local ». Voir [Variables Privées via my\(\) in the perlsub manpage](#) pour plus de détails.

`local()` modifie les variables listées pour qu'elles soient locales au bloc/fichier/eval englobant. Si plus d'une valeur est donnée, la liste doit être placée entre parenthèses. Voir [Valeurs Temporaires via local\(\) in the perlsub manpage](#) pour plus de détails, en particulier tout ce qui touche aux tables de hachage et aux tableaux liés (par `tie()`).

localtime EXPR

Convertit une date telle que retournée par la fonction `time` en un tableau de 9 éléments avec la date liée au fuseau horaire local. On l'utilise typiquement de la manière suivante :

```
# 0 1 2 3 4 5 6 7 8
($sec,$min,$heure,$mjour,$mois,$annee,$sjour,$ajour,$isdst) =
    localtime(time);
```

Tous les éléments du tableau sont numériques et restent tels qu'ils apparaissent dans la structure 'struct tm'. `$sec`, `$min` et `$heure` sont les secondes, les minutes et l'heure de l'instant spécifié. `$mjour` est le quantième du mois et `$mois` est le mois lui-même, dans l'intervalle 0..11 avec 0 pour janvier et 11 pour décembre. `$annee` est le nombre d'années depuis 1900 et donc `$annee` vaut 123 en l'an 2023. `$sjour` est le jour de la semaine avec 0 pour le dimanche et 3 pour le mercredi. `$ajour` est le jour de l'année dans l'intervalle 1..365 (ou 1..366 pour les années bissextiles.) `$isdst` est vrai si l'heure d'été est en cours à la date spécifiée et faux sinon.

Remarquez bien que `$annee` n'est pas que les deux derniers chiffres de l'année. Si vous supposez cela, vous créez des programmes non compatible an 2000 -- et vous ne voulez pas faire cela, n'est-ce pas ?

La bonne méthode pour obtenir une année complète sur 4 chiffres est tout simplement :

```
$annee += 1900;
```

Et pour obtenir les deux derniers chiffres de l'année (e.g. '01' en 2001):

```
$annee = sprintf("%02d", $annee % 100);
```

En l'absence de EXPR, `localtime()` utilise la date courante (`localtime(time)`).

Dans un contexte scalaire, retourne la valeur de `ctime(3)` :

```
$now_string = localtime; # e.g., "Thu Oct 13 04:54:34 1994"
```

La valeur scalaire n'est **pas** dépendante du locale, voir [the perllocale manpage](#), mais est construite de manière interne à Perl. Voir aussi le module `Time::Local` et les fonctions `strftime(3)` et `mktime(3)` du module POSIX. Pour obtenir quelque chose de similaire mais dont les chaînes de caractères des dates dépendent du locale, paramétrez vos variables d'environnement liées au locale (voir [the perllocale manpage](#)) et essayez par exemple :

```
use POSIX qw(strftime);
$now_string = strftime "%a %b %e %H:%M:%S %Y", localtime;
```

Notez que `%a` et `%b`, les formes courtes du jour de la semaine et du mois de l'année, n'ont pas obligatoirement 3 caractères de long.

lock TRUC

Cette fonction place un verrou coopératif sur une variable partagée ou sur l'objet référencé par *TRUC* pour la durée de la portée de ce lock.

lock() est un « mot clé faible » : cela signifie que si vous définissez une fonction par ce nom (avant tout appel à lock()), c'est cette fonction qui sera appelée. (En revanche, si vous dites `use threads` alors lock() est toujours un mot clé.) Voir *threads*.

log *EXPR*

log

Retourne le logarithme népérien ou naturel (base *e*) de *EXPR*. En l'absence de *EXPR*, retourne le log de `$_`. Pour obtenir le logarithme dans une autre base, utilisez la propriété suivante : le logarithme en base *N* d'un nombre est égal au logarithme naturel de ce nombre divisé par le logarithme naturel de *N*. Par exemple :

```
sub log10 {
    my $n = shift;
    return log($n)/log(10);
}
```

Voir exp pour l'opération inverse.

lstat *EXPR*

lstat

Fait la même chose que la fonction stat() (y compris de modifier le descripteur spécial `_`) mais fournit les données du lien symbolique au lieu de celles du fichier pointé par le lien. Si les liens symboliques ne sont pas implémentés dans votre système, un appel normal à stat() est effectué.

En l'absence de *EXPR*, utilise `$_`.

m//

L'opérateur de correspondance (d'expressions rationnelles). Voir the perlop manpage.

map *BLOC LISTE*

map *EXPR,LISTE*

Évalue le bloc *BLOC* ou l'expression *EXPR* pour chaque élément de *LISTE* (en affectant localement chaque élément à `$_`) et retourne la liste de valeurs constituée de tous les résultats de ces évaluations. L'évaluation du bloc *BLOC* ou de l'expression *EXPR* a lieu dans un contexte de liste si bien que chaque élément de *LISTE* peut produire zéro, un ou plusieurs éléments comme valeur retournée.

```
@chars = map(chr, @nums);
```

transcrit une liste de nombres vers les caractères correspondants. Et :

```
%hash = map { getkey($_) => $_ } @array;
```

est juste une manière rigolote de dire :

```
%hash = ();
foreach $_ (@array) {
    $hash{getkey($_)} = $_;
}
```

Remarquez que du fait que `$_` est une référence vers la liste de valeurs, il peut être utilisé pour modifier les éléments du tableau. Bien que cela soit pratique et supporté, cela peut produire des résultats bizarres si *LISTE* n'est pas un tableau nommé. L'utilisation d'une boucle `foreach` est plus claire dans de nombreux cas. Voir aussi grep pour produire un tableau composé de tous les éléments de la liste originale pour lesquels *BLOC* ou *EXPR* est évalué à vrai (`true`).

mkdir *FILENAME, MASK*

mkdir

Crée le répertoire dont le nom est spécifié par FILENAME avec les droits d'accès spécifiés par MASK (et modifiés par umask). En cas de succès, retourne TRUE (vrai). Sinon, retourne false (faux) et positionne la variable \$! (errno). Par défaut, MASK vaut 0777.

En général, il vaut mieux créer des répertoires avec un MASK permissif et laisser l'utilisateur modifier cela via son umask que de fournir un MASK trop restrictif ne permettant pas à l'utilisateur d'être plus permissif. L'exception à cette règle concerne les répertoires ou les fichiers doivent être privés (fichiers de messagerie par exemple). umask discute plus en détails du choix de MASK.

Notez que, selon la norme POSIX 1003.1–1996, FILENAME peut se terminer par zéro, un ou plusieurs \/. Certains systèmes d'exploitation ou de fichiers ne le permettent pas. Donc, pour que tout le monde soit content. Perl enlève automatiquement ces caractères \/. finaux.

msgctl ID,CMD,ARG

Appelle la fonction msgctl(2) des IPC System V. Vous devrez probablement dire :

```
use IPC::SysV;
```

au préalable pour avoir les définitions correctes des constantes. Si CMD est IPC_STAT alors ARG doit être une variable qui pourra contenir la structure `msgid_ds` retournée.

Renvoie la même chose que ioctl() : la valeur undef en cas d'erreur, ``0 but true" pour la valeur zéro ou la véritable valeur dans les autres cas. Voir aussi la documentation de IPC::SysV et IPC::Semaphore::Msg.

msgget KEY,FLAGS

Appelle la fonction msgget(2) des IPC System V. Retourne l'id de la queue de messages ou la valeur undef en cas d'erreur. Voir aussi IPC::SysV et IPC::SysV::Msg.

msgrcv ID,VAR,SIZE,TYPE,FLAGS

Appelle la fonction `msgrcv` des IPC System V pour recevoir un message à partir de la queue de message d'identificateur ID et le stocker dans la variable VAR avec une taille maximale de SIZE. Remarquez que, si un message est reçu, le type de message sera la première chose dans VAR et que la taille maximale de VAR est SIZE plus la taille du type de message. Ces données compactées peuvent être décompactées par unpack("l! a*"). Souille la variable VAR. Retourne TRUE (vrai) en cas de succès ou false (faux) sinon. Voir aussi IPC::SysV et IPC::SysV::Msg.

msgsnd ID,MSG,FLAGS

Appelle la fonction `msgsnd` des IPC System V pour envoyer le message MSG dans la queue de messages d'identificateur ID. MSG doit commencer par l'entier long natif donnant le type de message suivi de la longueur réelle du message suivi du message lui-même. Cette donnée compactée peut être produite par pack("l! a*", \$type, \$message).

Retourne true (vrai) en cas de succès ou false (faux) en cas d'erreur. Voir aussi IPC::SysV et IPC::SysV::Msg.

my EXPR

my TYPE EXPR

my EXPR : ATTRIBUTES

my TYPE EXPR : ATTRIBUTES

my() déclare les variables listées comme étant locales (lexicalement) au bloc, fichier ou eval() englobant. Si plus d'une variable est listée, la liste doit être placée entre parenthèses.

La sémantique exacte et l'interface avec TYPE et ATTRIBUTES est encore en cours d'évolution. TYPE est actuellement lié à l'utilisation de la directive `fields` et les attributs sont gérés par la directive `attributes` ou, depuis la version 5.8.0 de Perl, via le module `Attribute::Handlers`. Voir [Variables Privées via my\(\) in the perlsub manpage](#) pour plus de détails ainsi que `fields`, `attributes` et [the Attribute::Handlers manpage](#).

next LABEL

next

La commande `next` fonctionne comme l'instruction `continue` du C ; elle commence la prochaine itération d'une boucle :

```
LINE: while (<STDIN>) {
    next LINE if /^#/;      # ne pas traiter les commentaires
    #...
}
```

Si il y avait un bloc `continue` dans cet exemple, il serait exécuté même pour les lignes ignorées. Si LABEL est omis, la commande se réfère au bloc englobant le plus intérieur. `next` ne peut pas être utilisé pour sortir d'un bloc qui doit retourner une valeur comme `eval {}`, `sub {}` ou `do {}` et ne devrait pas être utilisé pour sortir d'une opération `grep()` ou `map()`.

Voir aussi `continue` pour voir comment `last`, `next` et `redo` fonctionne.

no Module VERSION LISTE

no Module VERSION

no Module LISTE

no Module

Voir la fonction `use` pour laquelle `no` est le contraire.

oct EXPR

oct

Interprète EXPR comme une chaîne octale et retourne la valeur correspondante. (Si il s'avère que EXPR commence par `0x`, elle sera interprétée comme un chaîne hexadécimale. Si EXPR commence par `0b`, elle sera interprétée comme une chaîne binaire. Dans tous les cas, les blancs initiaux sont ignorés) La ligne suivante manipule les chaîne décimales, binaires, octales et hexadécimales comme le fait la notation standard Perl ou C :

```
$val = oct($val) if $val =~ /^0/;
```

Si EXPR est absent, la commande s'applique à `$_`. Pour réaliser l'opération inverse (produire la représentation octale d'un nombre), utilisez `sprintf()` ou `printf()` :

```
$perms = (stat("filename"))[2] & 07777;
$oct_perms = sprintf "%lo", $perms;
```

La fonction `oct()` est couramment utilisée pour convertir une chaîne telle que `644` en un mode d'accès pour fichier par exemple. (perl convertit automatiquement les chaînes en nombres si besoin mais en supposant qu'ils sont en base 10.)

open DESCRIPTEUR,EXPR

open DESCRIPTEUR,MODE,EXPR

open DESCRIPTEUR,MODE,EXPR,LISTE

open DESCRIPTEUR,MODE,REFERENCE

open DESCRIPTEUR

Ouvre le fichier dont le nom est donné par EXPR et l'associe à DESCRIPTEUR.

(La suite de cette section est une description exhaustive de `open()`. Pour une introduction plus douce, vous devriez regarder [perlopentut](#).)

Si DESCRIPTEUR est une variable lexicale (`my`) indéfinie, cette variable recevra une référence vers un nouveau descripteur anonyme. Sinon, si DESCRIPTEUR est une expression, sa valeur est utilisée en tant que nom du descripteur à associer. (Ceci est considéré comme une référence symbolique donc `use strict 'refs'` ne devrait pas être actif.)

Si EXPR est omis, la variable scalaire du même nom que le DESCRIPTEUR contient le nom du fichier. (Remarquez que les variables lexicales -- celles déclarées par `my()` -- ne fonctionnent pas dans ce cas ; donc si vous utilisez `my()`, spécifiez EXPR dans votre appel à `open`.)

Si au moins trois arguments sont spécifiés alors le mode d'ouverture et le nom du fichier sont séparés. Si MODE est '`<`' ou rien du tout, le fichier est ouvert en lecture. Si MODE est '`>`', le fichier est tronqué puis ouvert en écriture en étant créé si nécessaire. Si MODE est '`>>`', le fichier est ouvert en écriture et en mode ajout. Là encore, il sera créé si nécessaire. Vous pouvez ajouter un '+' devant '`>`' ou '`<`' pour indiquer que vous voulez à la fois les droits d'écriture et de lecture sur le fichier ; Ceci étant '`+<`' est toujours mieux pour les mises à jour en lecture/écriture -- le mode '`+>`' écraserait le fichier au préalable.

Habituellement, il n'est pas possible d'utiliser le mode lecture/écriture pour des fichiers textes puisqu'ils ont des tailles d'enregistrements variables. Voir l'option `-i` dans [the perlrun manpage](#) pour une meilleure approche. Le fichier est créé avec les droits `0666` modifiés par la valeur de `umask` du processus courant.

Ces différents préfixes correspondent aux différents modes d'ouverture de `fopen(3)` : '`r`', '`r+`', '`w`', '`w+`', '`a`' et '`a+`'.

Dans sa forme à 1 ou 2 arguments, le mode et le nom de fichier peuvent être concaténés (dans cet ordre), éventuellement séparés par des espaces. Il est possible d'omettre le mode si c'est '`<`'.

Si le nom de fichier commence par '`|`', le nom de fichier est interprété comme une commande vers laquelle seront dirigées les sorties (via un tube -- en anglais pipe) et si le nom de fichier se termine par '`|`', le nom de fichier est interprété comme une commande dont la sortie sera récupérée (via un tube -- en anglais pipe). Voir [Utilisation de open\(\) pour la CIP in the perlipc manpage](#) pour des exemples à ce sujet. (Vous ne pouvez pas utiliser `open()` pour une commande qui utiliserait un même tube à la fois pour ses entrées et pour ses sorties mais [the IPC::Open2 manpage](#), [the IPC::Open3 manpage](#) et [Communication Bidirectionnelle avec un autre Processus in the perlipc manpage](#) proposent des solutions de remplacement.)

Pour les appels avec au moins trois arguments, si MODE est '`|-`', le nom de fichier est interprété comme une commande vers laquelle seront dirigées les sorties et si le MODE est '`-|`', le nom de fichier sera interprété comme une commande dont la sortie sera récupérée (via un tube -- en anglais pipe). Pour retomber sur une forme à deux arguments, il suffit de remplacer le moins ('`-`') par la commande elle-même. Voir [Utilisation de open\(\) pour la CIP in the perlipc manpage](#) pour des exemples à ce sujet. (Vous ne pouvez pas utiliser `open()` pour une commande qui utiliserait un même tube à la fois pour ses entrées et pour ses sorties mais [the IPC::Open2 manpage](#), [the IPC::Open3 manpage](#) et [Communication Bidirectionnelle avec un autre Processus in the perlipc manpage](#) proposent des solutions de remplacement.)

Pour les appels avec au moins trois arguments et faisant au mécanisme de tube, si LISTE est spécifiée (ce sont les arguments après la commande) alors LISTE devient les arguments de la commande invoquée si la plateforme l'accepte. La sémantique de `open` avec plus de trois arguments pour un mode sans tube n'est pas encore spécifiée. Des ``filtres'' expérimentaux peuvent donner un sens à ces arguments supplémentaires.

Dans la forme à un ou deux arguments, ouvrir '`-`' revient à ouvrir STDIN tandis qu'ouvrir '`>`' revient à ouvrir STDOUT.

Vous pouvez utiliser la forme à trois arguments en spécifiant des ``filtres" d'entrée/sortie (anciennement appelés ``disciplines") à appliquer au descripteur afin de modifier la manière dont sont traités les entrées et les sorties (Voir [open](#) et *PerlIO* pour plus d'informations). Par exemple :

```
open(FH, "<:utf8", "file")
```

ouvrira le fichier ``file" encodé en UTF-8 et contenant des caractères Unicode. Voir *perluniintro*. (Remarquez que si des filtres sont spécifiés dans un appel à au moins trois arguments alors les filtres par défaut fixés via la directive [open](#) sont ignorés.)

Open renvoie une valeur non nulle en cas de succès et undef sinon. Si [open\(\)](#) utilise un tube, la valeur de retour sera le PID du sous-processus.

Si vous utilisez Perl sur un système qui fait une distinction entre les fichiers textes et les fichiers binaires alors vous devriez regarder du côté de [binmode](#) pour connaître les astuces à ce sujet. La principale différence entre les systèmes nécessitant [binmode\(\)](#) et les autres réside dans le format de leurs fichiers textes. Des systèmes tels que Unix, MacOS et Plan9 qui délimitent leurs lignes par un seul caractère et qui encodent ce caractère en C par "\n" ne nécessitent pas [binmode\(\)](#). Les autres en ont besoin.

À l'ouverture d'un fichier, c'est généralement une mauvaise idée de continuer l'exécution normale si la requête échoue, ce qui explique pourquoi [open\(\)](#) est si fréquemment utilisé en association avec [die\(\)](#). Même si [die\(\)](#) n'est pas ce que voulez faire (par exemple dans un script CGI où vous voulez récupérer un message d'erreur joliment présenté (il y a des modules qui peuvent vous aider pour cela)), vous devriez toujours vérifier la valeur retournée par l'ouverture du fichier. L'une des rares exceptions est lorsque vous voulez travailler sur un descripteur non ouvert.

Un appel sous la forme de trois arguments en mode lecture/écriture avec un troisième argument indéfini ([undef](#)) est un cas spécial :

```
open(TMP, "+>", undef) or die ...
```

ouvrira un descripteur vers un fichier temporaire anonyme.

Les descripteurs peuvent être ouvert sur des fichiers ``en mémoire" stockés dans des scalaires Perl via :

```
open($fh, '>', \ $variable) || ...
```

Par contre, si vous essayez de ré-ouvrir STDOUT ou STDERR comme des fichiers ``en mémoire", vous devez les fermer auparavant :

```
close STDOUT;
open STDOUT, '>', \ $variable or die "Can't open STDOUT: $!";
```

Exemples :

```
$ARTICLE = 100;
open ARTICLE or die "Can't find article $ARTICLE: $!\n";
while (<ARTICLE>) {...
open(LOG, '>>/usr/spool/news/twitlog'); # (log is reserved)
# si le open échoue, les sorties sont perdues
open(DBASE, '+<', 'dbase.mine')          # ouverture pour mise à jour
  or die "Can't open 'dbase.mine' for update: $!";
open(DBASE, '+<dbase.mine')             # idem
  or die "Can't open 'dbase.mine' for update: $!";
open(ARTICLE, '-|', "caesar <$article")  # décryptage de l'article
  or die "Can't start caesar: $!";
open(ARTICLE, "caesar <$article |")      # idem
  or die "Can't start caesar: $!";
open(EXTRACT, "|sort >/tmp/Tmp$$")      # $$ est notre ID de process
  or die "Can't start sort: $!";
# fichiers en mémoire
open(MEMORY, '>', \ $var)
  or die "Can't open memory file: $!";
print MEMORY "foo!\n";                   # les sorties seront envoyées vers $var
# traitement de la liste des fichiers fournie en argument
```

perlfunc – Fonctions Perl prédéfinies

```
foreach $file (@ARGV) {
    process($file, 'fh00');
}
sub process {
    my($filename, $input) = @_;
    $input++; # c'est une incrémentation de chaîne
    unless (open($input, $filename)) {
        print STDERR "Can't open $filename: $!\n";
        return;
    }
    local $_;
    while (<$input>) { # remarquez l'utilisation de l'indirection
        if (/^#include "(.*)"/) {
            process($1, $input);
            next;
        }
        #... # ce qu'il faut faire...
    }
}
```

Vous pouvez aussi, dans la tradition du Bourne shell, spécifier une expression `EXPR` commençant par `'>&'` auquel cas le reste de la chaîne sera interprété comme le nom d'un descripteur (ou son numéro si c'est une valeur numérique) à dupliquer puis à ouvrir. Vous pouvez utiliser `&` après `>`, `>>`, `<`, `+>`, `+>>` et `+<`. Le mode que vous spécifiez devrait correspondre à celui du descripteur original. (La duplication d'un descripteur ne prend pas en compte l'éventuel contenu des tampons d'entrées/sorties). Si vous utilisez l'appel avec 3 arguments alors vous devez fournir soit un nombre, soit le nom du descripteur soit une référence globale ``normale''.

Voici un script qui sauvegarde, redirige et restaure `STDOUT` et `STDERR` de différentes manières :

```
#!/usr/bin/perl
open my $oldout, ">&STDOUT" or die "Can't dup STDOUT: $!";
open OLDERR, ">&", \*STDERR or die "Can't dup STDERR: $!";
open STDOUT, '>', "foo.out" or die "Can't redirect STDOUT: $!";
open STDERR, ">&STDOUT" or die "Can't dup STDOUT: $!";
select STDERR; $| = 1; # make unbuffered
select STDOUT; $| = 1; # make unbuffered
print STDOUT "stdout 1\n"; # this works for
print STDERR "stderr 1\n"; # subprocesses too
close STDOUT;
close STDERR;
open STDOUT, ">&", $oldout or die "Can't dup \">$oldout: $!";
open STDERR, ">&OLDERR" or die "Can't dup OLDERR: $!";
print STDOUT "stdout 2\n";
print STDERR "stderr 2\n";
```

Si vous spécifiez `'<&=N'`, où `N` est un nombre, alors Perl fera l'équivalent d'un `fdopen()` en C sur ce descripteur de fichier ; c'est plus économe en descripteur de fichier. Par exemple :

```
open(DESCRIPTEUR, "<&=$fd")
```

ou

```
open(DESCRIPTEUR, "<&=", $fd)
```

Remarquez bien que cette fonctionnalité dépend la fonction `fdopen()` de votre bibliothèque C. Sur la plupart des systèmes UNIX, `fdopen()` est connu pour échouer lorsque le nombre de descripteurs de fichiers dépasse une certaine limite, typiquement 255. Si vous avez besoin de descripteurs de fichiers plus nombreux, vous devriez examiner la possibilité de recompiler Perl en utilisant la bibliothèque `PerlIO`.

Vous pouvez savoir si Perl a été compilé avec `PerlIO` ou non en exécutant `perl -V` et en regardant la ligne `useperlio=`. Si `useperlio` est défini, vous avez `PerlIO`.

Si vous ouvrez un tube (pipe) sur la commande '-', i.e. soit '|-' soit '-|', alors vous faites un fork implicite et la valeur de retour de open est le PID du processus fils pour le processus père et 0 pour le processus fils. (Utilisez `defined($pid)` pour savoir si le open s'est bien déroulé.) Le descripteur se comporte normalement pour le père mais pour le processus fils, les entrées/sorties sur ce descripteur se font via STDIN/STDOUT. Dans le processus fils, le descripteur n'est pas ouvert -- les échanges se font via les nouveaux STDIN ou STDOUT. Cela s'avère utile lorsque vous voulez avoir un contrôle plus fin sur la commande exécutée par exemple lorsque vous avez un script setuid et que vous ne voulez pas que le shell traite les méta-caractères dans les commandes. Les lignes suivantes sont quasiment équivalentes (trois à trois) :

```
open(FOO, "|tr '[a-z]' '[A-Z]'");
open(FOO, '|-', "tr '[a-z]' '[A-Z]'");
open(FOO, "|-" || exec 'tr', '[a-z]', '[A-Z]');
open(FOO, '|-', "tr", '[a-z]', '[A-Z]');
open(FOO, "cat -n '$file'|");
open(FOO, '-|', "cat -n '$file'");
open(FOO, "-|" || exec 'cat', '-n', $file);
open(FOO, '-|', "cat", '-n', $file);
```

Le dernier exemple de chacun des deux blocs montre l'utilisation des tubes avec le passage de paramètres. Ce n'est pas disponible sur toutes les plateformes. Si votre plateforme propose un vrai `fork()` (en d'autres termes, si vous êtes sur UNIX) alors vous pouvez utiliser cette syntaxe.

Voir [Ouvertures Sûres d'un Tube in the perlipc manpage](#) pour plus d'exemples à ce sujet.

À partir de la version v5.6.0, Perl tente de vider les tampons de tous les fichiers ouverts en écriture avant toute opération impliquant un fork mais ce n'est pas supporté sur toutes les plates-formes (voir [the perlport manpage](#)). Pour être plus sûr, vous devriez positionner la variable `$|` (`$AUTOFLUSH` en anglais) ou appelé la méthode `autoflush()` des objets `IO::Handle` pour chacun des descripteurs ouverts afin d'éviter toute perte de données. Sur les systèmes qui supportent le drapeau fermeture-à-l-exécution (`close-on-exec`) sur les fichiers, ce drapeau sera positionné pour de nouveaux descripteurs de fichier en fonction de la valeur de `$^F`. Voir [\\$^F in the perlvar manpage](#).

La fermeture d'un descripteur utilisant un tube (pipe) amène le processus père à attendre que son fils se termine puis à retourner la valeur de statut dans `$?`.

Le nom de fichier passé à open dans la forme à 1 ou 2 arguments verra ses éventuels espaces avant et après supprimés et les caractères de redirection normaux seront respectés. Cette fonctionnalité, connue sous le nom de "ouverture magique", autorise plein de bonnes choses. Un utilisateur peut spécifier un nom de fichier tel que ```rsh cat file |``` ou alors vous pouvez modifier certains noms de fichiers selon vos besoins :

```
$filename =~ s/(.*\.gz)\s*$/gzip -dc < $1|/;
open(FH, $filename) or die "Can't open $filename: $!";
```

Utilisez la forme à 3 arguments pour ouvrir un fichier dont le nom contient des caractères quelconques :

```
open(FOO, '<', $file);
```

sinon il est nécessaire de protéger les caractères spéciaux et tous les espaces avant et/ou après :

```
$file =~ s#^\s#./$1#;
open(FOO, "< $file\0");
```

(cela peut ne pas fonctionner sur certains systèmes de fichier bizarres). Vous devez choisir consciemment entre la forme magique ou la forme à 3 arguments de open() :

```
open IN, $ARGV[0];
```

autorisera l'utilisateur à spécifier un argument de la forme ```rsh cat file |``` mais ne marchera pas avec un nom de fichier contenant des espaces alors que :

```
open IN, '<', $ARGV[0];
```

a exactement les limitation inverses.

Si vous voulez un ``vrai" `open()` à la C (voir `open(2)` sur votre système) alors vous devriez utiliser la fonction `sysopen()` qui ne fait rien de magique. C'est un autre moyen de protéger vos noms de fichiers de toute interprétation. Par exemple :

```
use IO::Handle;
sysopen(HANDLE, $path, O_RDWR|O_CREAT|O_EXCL)
    or die "sysopen $path: $!";
$oldfh = select(HANDLE); $| = 1; select($oldfh);
print HANDLE "stuff $$\n";
seek(HANDLE, 0, 0);
print "File contains: ", <HANDLE>;
```

En utilisant le constructeur du paquetage `IO::Handle` (ou de l'une de ses sous-classes telles que `IO::File` ou `IO::Socket`), vous pouvez générer des descripteurs anonymes qui ont la même portée que les variables qui gardent une référence sur eux et qui se ferment automatiquement dès qu'ils sont hors de portée :

```
use IO::File;
#...
sub read_myfile_munged {
    my $ALL = shift;
    my $handle = new IO::File;
    open($handle, "myfile") or die "myfile: $!";
    $first = <$handle>
        or return ();      # Fermeture automatique ici
    mung $first or die "mung failed";      # Ou ici.
    return $first, <$handle> if $ALL;      # Ou ici.
    $first;                  # Ou ici.
}
```

Voir `seek()` pour de plus amples informations sur le mélange entre lecture et écriture.

opendir DIRHANDLE,EXPR

Ouvre le répertoire nommé par `EXPR` pour un traitement par `readdir()`, `tellmdir()`, `seekdir()`, `rewinddir()` et `closedir()`. Retourne true (vrai) en cas de succès. Les `DIRHANDLES` ont leur propre espace de noms séparé de celui des `DESCRIPTEURS`.

ord EXPR

ord

Retourne la valeur numérique (dans l'encodage 8-bit natif comme ASCII, EBCDIC ou Unicode) du premier caractère de `EXPR`. Si `EXPR` est omis, utilise `$_`.

Voir `chr` pour l'opération inverse. Voir `perlunicode` et `encoding` pour en savoir plus sur Unicode.

our EXPR

our TYPE EXPR

our EXPR : ATTRS

our TYPE EXPR : ATTRS

Un `our` déclare les variables listées comme étant des variables globales valides dans la portée d'un bloc englobant, d'un fichier ou d'un `eval`. Il a donc la même portée qu'une déclaration ``my" mais ne crée pas de variable locale. Si plus d'une valeur est listée, la liste doit être placée entre parenthèses. La déclaration `our` n'a aucun effet sémantique à moins que ``use strict vars" soit actif auquel cas, cela vous permet d'utiliser les variables globales déclarées sans les qualifier par un nom de package. (Mais seulement dans la portée lexicale de votre déclaration `our`. En cela, il diffère de ``use vars" dont la portée est globale au package.)

Une déclaration `our` déclare un variable globale qui sera visible dans toute la portée lexicale de cette déclaration, même entre différents packages. Le package d'appartenance de la variable est déterminée lors de la déclaration et non lors de l'utilisation. Cela signifie que vous aurez les comportements suivants :

```
package Foo;
our $bar;          # déclare $Foo::bar
$bar = 20;
package Bar;
print $bar;       # affiche 20 (valeur de $Foo::bar)
```

De multiples déclarations `our` sont autorisées dans la même portée lexicale si elles sont dans des packages différents. Si elles sont dans le même package, Perl produira un avertissement si vous le lui avez demandé.

```
use warnings;
package Foo;
our $bar;          # déclare $Foo::bar
$bar = 20;
package Bar;
our $bar = 30;     # déclare $Bar::bar
print $bar;       # affiche 30
our $bar;          # émission d'un avertissement
```

Une déclaration `our` peut aussi être associée à une liste d'attributs.

La sémantique exacte et l'interface de `TYPE` et `ATTRS` sont encore en cours d'évolution.

`TYPE` est actuellement lié à l'utilisation de la directive `fields` et les attributs (`ATTRS`) sont gérés par la directive `attributes` ou, depuis perl 5.8.0, via le module `Attribute::Handlers`. Voir [Variables Privées via my\(\) in the perlsub manpage](#) pour plus de détails ainsi que `fields`, `attributes` et [the Attribute::Handlers manpage](#).

Actuellement, le seul attribut reconnu par `our()` est `unique` qui indique qu'une seule et unique instance de la variable globale doit être utilisée par tous les interpréteurs que lance le programme dans un environnement d'interpréteurs multiples. (Le comportement par défaut est que chaque interpréteur possède sa propre instance.) Exemples :

```
our @EXPORT : unique = qw(foo);
our %EXPORT_TAGS : unique = (bar => [qw(aa bb cc)]);
our $VERSION : unique = "1.00";
```

Notez aussi que cette attribut a pour effet de rendre la variable globale non-modifiable dès que le premier interpréteur est cloné (par exemple lorsque le premier fil d'exécution est créé).

Un environnement d'interpréteurs multiples peut aussi provenir de l'émulation de `fork()` sur les plateformes Windows ou d'un interpréteur embarqué dans une application multi-fils (multi-threaded application). L'attribut `unique` en fait rien dans tout autre environnement.

pack TEMPLATE,LISTE

Prend une liste de valeurs et les transforme en une chaîne en utilisant les règles décrites par `TEMPLATE`. La chaîne résultante est la concaténation des valeurs converties.

Typiquement, chaque valeur convertie aura sa représentation du niveau machine. Par exemple, sur une machine 32-bit, un entier converti sera représenté par une séquence de 4 octets.

Le `TEMPLATE` est une séquence de caractères qui donne l'ordre et le type des valeurs, de la manière suivante :

- a Une chaîne contenant des données binaires quelconques, éventuellement complétée par des caractères NUL.
- A Un texte (ASCII), complétée par des blancs.
- Z Une chaîne (ASCII) terminée par un caractère NUL, complétée par des caractères NUL.
- b Une chaîne de bits (en ordre croissant dans chaque octet, comme pour `vec()`).

perlfunc – Fonctions Perl prédéfinies

- B Une chaîne de bits (en ordre décroissant dans chaque octet).
- h Une chaîne hexadécimale (chiffres hexadécimaux faibles en premier).
- H Une chaîne hexadécimale (chiffres hexadécimaux forts en premier).
- c La valeur d'un caractère signé.
- C La valeur d'un caractère non signé. Ne traite que des octets.
Voir U pour Unicode.
- s La valeur d'un entier court (short) signé
- S La valeur d'un entier court (short) non signé
(Ce 'short' est `_exactement_` sur 16 bits ce qui peut être différent de ce qu'un compilateur C local appelle 'short'. Si vous voulez un short natif, utilisez le suffixe '!'.)
- i La valeur d'un entier (integer) signé.
- I La valeur d'un entier (integer) non signé.
(Cet 'integer' est `_au moins_` sur 32 bits. Sa taille exact dépend de ce que le compilateur C local appelle 'int' et peut même être plus long que le 'long' décrit à l'item suivant.)
- l La valeur d'un entier long (long) signé.
- L La valeur d'un entier long (long) non signé.
(Ce 'long' est `_exactement_` sur 32 bits ce qui peut être différent de ce qu'un compilateur C local appelle 'long'. Si vous voulez un long natif, utilisez le suffixe '!'.)
- n Un entier court non signé (short) dans l'ordre "réseau" (big-endian).
- N Un entier long non signé (long) dans l'ordre "réseau" (big-endian).
- v Un entier court non signé (short) dans l'ordre "VAX" (little-endian).
- V Un entier long non signé (long) dans l'ordre "VAX" (little-endian).
(Ces 'shorts' et ces 'longs' font `_exactement_` 16 et 32 bits respectivement.)
- q Une valeur quad (64-bit) signée.
- Q Une valeur quad non signée.
(Les quads ne sont disponibles que si votre système supporte les entiers 64-bit `_et_` que si Perl a été compilé pour les accepter. Provoquera une erreur fatale sinon.)
- j Une valeur entière signée (un entier interne de Perl, IV)
- J Une valeur entière non-signée (un entier non-signé de Perl, UV)
- f Un flottant simple précision au format natif.
- d Un flottant double précision au format natif.
- F Une valeur à virgule flottante en format natif
(un flottant Perl interne, NV).
- D Un flottant long en double précision en format natif
(Ce format doit être connue de votre système `_ET_` Perl doit être compilé pour les reconnaître.
Produit une erreur fatale sinon.)
- p Un pointeur vers une chaîne terminée par un caractère NUL.
- P Un pointeur vers une structure (une chaîne de longueur fixe).
- u Une chaîne uuencodée.
- U Un nombre d'un caractère Unicode. Encode en UTF-8 en interne
(ou en UTF-EBCDIC sur les plateformes EBCDIC).
- w Un entier BER compressé. Ses octets représentent chacun un entier non signé en base 128. Les chiffres les plus significatifs viennent en premier et il y a le moins de chiffres possibles. Le huitième bit (le bit de poids fort) est toujours à 1 pour chaque octets sauf le dernier.
- x Un octet nul.
- X Retour en arrière d'un octet.
- @ Remplissage par des octets nuls jusqu'à une position absolue.
(Début d'un ()-groupe.

Les règles suivantes s'appliquent :



perlfunc – Fonctions Perl prédéfinies

Chaque lettre peut éventuellement être suivie d'un nombre spécifiant le nombre de répétitions. Pour tous les types sauf `a`, `A`, `Z`, `b`, `B`, `h`, `H`, `@`, `x`, `X` et `P`, la fonction `pack` utilisera autant de valeurs de la liste `LISTE` que nécessaire. Une `*` comme valeur de répétition demande à utiliser toutes les valeurs restantes excepté pour `@`, `x` et `X` où c'est équivalent à 0 et pour `u` où c'est équivalent à 1 (ou 45, ce qui est la même chose). Le nombre de répétitions peut éventuellement être entouré de crochets comme dans `pack 'C[80]', @arr`.

Il est possible de remplacer ce nombre de répétitions par un template placé entre crochets. C'est alors la longueur en octets de ce template compacté qui est utilisée comme nombre de répétitions. Par exemple, `x[L]` sautera autant d'octets que le nombre d'octets composant un long ; Le template `$x X[$t] $t` pourra décompacter deux fois ce que décompactera `$t`. Si le template entre crochets contient des commandes d'alignement (comme dans `x![d]`), sa longueur compactée sera calculée en supposant que le début du template est à l'alignement maximal.

Utilisé avec `Z`, `*` déclenchera l'ajout d'un octet nul final (donc la valeur compactée sera d'un octet plus longue que la longueur (`length`) de la valeur initiale).

La valeur de répétition pour `u` est interprétée comme le nombre maximale d'octets à encoder par ligne produite avec 0 et 1 remplacé par 45.

- ◆ Les types `a`, `A` et `Z` n'utilisent qu'une seule valeur mais la compacte sur la longueur spécifiée en la complétant éventuellement par des espaces ou des caractères NUL. Lors du décompactage, `A` supprime les espaces et les caractères NUL finaux, `Z` supprime tout ce qui suit le premier caractère NUL alors que `a` laisse la valeur complète. Lors du compactage, `a` et `Z` sont équivalents.

Si la valeur à compacter est trop longue, elle est tronquée. Si elle est trop longue et qu'une longueur explicite `$count` est fournie, `Z` compactera uniquement `$count-1` octets suivi d'un octet nul. Donc, `Z` compacte un caractère nul final en toutes circonstances.

- ◆ De la même manière, les types `b` et `B` remplissent la chaîne compactée avec autant de bits que demandés. Chaque octet du champ d'entrée de `pack()` produira 1 bit dans le résultat. Chaque bit résultant est le bit le moins significatif de l'octet d'entrée (i.e. `ord($octet)%2`). En particulier, les octets "0" et "1" produisent les bits 0 et 1 exactement comme le font "\0" et "\1".

En partant du début de la chaîne d'entrée de `pack()`, chaque 8-uplet d'octets est converti en un octet de sortie. Avec le format `b`, le premier octet du 8-uplet détermine le bit le moins significatif de l'octet alors qu'avec le format `B`, il détermine le bit le plus significatif.

Si la longueur de la chaîne d'entrée n'est pas exactement divisible par 8, le reste est compacté comme si la chaîne d'entrée était complétée par des octets nuls à la fin. De manière similaire, lors du décompactage, les bits "supplémentaires" sont ignorés.

perlfunc – Fonctions Perl prédéfinies

Si la chaîne d'entrée de `pack()` est plus longue que nécessaire, les octets en trop sont ignorés. Une valeur de répétition de `*` demande à utiliser tous les octets du champ d'entrée. Lors du décompactage, les bits sont convertis en une chaîne de "0" et de "1".



`h` et `H` compactent une chaîne hexadécimale (par groupe de 4-bit représentant un chiffre hexadécimale 0–9a–f).

Chaque octet du champ d'entrée de `pack()` génère 4 bits du résultat. Pour les octets non alphabétiques, le résultat est basé sur les 4 bits les moins significatifs de l'octet considéré (i.e. sur `ord($octet)%16`). En particulier, les octets "0" et "1" génèrent 0 et 1 comme le font les octets "\0" et "\1". Pour les octets "a" . . "f" et "A" . . "F", le résultat est compatible avec les chiffres hexadécimaux habituels et donc "a" et "A" génèrent tous les deux le groupe de 4 bits `0xa=10`. Le résultat pour les octets "g" . . "z" et "G" . . "Z" n'est pas clairement défini.

En partant du début de la chaîne d'entrée de `pack()`, chaque paire d'octets est convertie en 1 octet de sortie. Avec le format `h`, le premier octet de la paire détermine les 4 bits les moins significatifs de l'octet résultant alors qu'avec le format `H`, il détermine les 4 bits les plus significatifs.

Si la longueur de la chaîne d'entrée n'est pas paire, `pack()` se comportera comme si un octet nul avait été ajouté à la fin. De manière similaire, lors du décompactage (par `unpack()`), les groupes de 4 bits supplémentaires sont ignorés.

Si la chaîne d'entrée de `pack()` est plus longue que nécessaire, les octets supplémentaires sont ignorés. Une valeur de répétition de `*` demande à utiliser tous les octets de la chaîne d'entrée. Lors du décompactage, les bits sont convertis en une chaîne de chiffres hexadécimaux.



Le type `"p"` compactera une chaîne terminée par un caractère NUL dont on donne le pointeur. Il est de votre responsabilité de vous assurer que la chaîne n'est pas une valeur temporaire (qui pourrait être désallouée avant l'utilisation de la valeur compactée). Le type `P` compactera une structure de la taille indiquée par la longueur. Un pointeur NULL est créé si la valeur correspondant à `"p"` ou `"P"` est undef. Idem pour `unpack()`.



Le caractère `/` dans `TEMPLATE` autorise le compactage et le décompactage de chaîne donc la représentation compacte est la longueur en octets suivie de la chaîne elle-même. Vous écrivez *longueur-item / chaîne-item*.

La *longueur-item* peut être n'importe quelle lettre d'un template `pack` et elle représente comment cette valeur de longueur est compactée. Les plus utilisées sont les longueurs entières comme `n` (pour des chaînes Java), `w` (pour ASN.1 ou SNMP) et `N` (pour les XDR de Sun).

longueur-item peut être n'importe quel type de lettre d'un `TEMPLATE pack` et décrit la manière dont la longueur est compactée. Les plus courantes sont `n` (pour les

chaînes Java), *w* (pour ASN.1 ou SNMP) et *N* (pour les XDR de SUN).

chaîne-item doit être, s'il est présent, "A*", "a*" ou "Z*". Pour le décompactage (via `unpack()`), la longueur de la chaîne est celle obtenue par *longueur-item* mais si vous spécifiez '*', elle sera ignorée.

```
unpack 'C/a', "\04Gurusamy";           donne 'Guru'
unpack 'a3/A* A*', '007 Bond J ';     donne (' Bond', 'J')
pack 'n/a* w/a*', 'hello, ', 'world'; donne "\000\006hello,\005world"
```

longueur-item n'est pas retourné explicitement par `unpack()`.

L'ajout d'un compteur de répétition à la lettre *longueur-item* est inutile sauf si cette lettre est A, a ou Z. Le compactage avec une *longueur-item* spécifiée par a ou z peut introduire des caractères "\000" que Perl ne considérera pas comme légal dans une chaîne numérique.



Les types entiers *s*, *S*, *l* et *L* peuvent être immédiatement suivi d'un suffixe ! pour indiquer des shorts ou des longs natifs -- comme vous pourrez le voir dans l'exemple suivant, *l* ne signifie pas exactement 32 bits puisque le long natif (tel qu'il est vu par le compilateur C natif) peut être plus long. C'est une préoccupation principalement sur des plates-formes 64-bit. Vous pouvez voir si l'utilisation de ! fait une différence en faisant :

```
print length(pack("s")), " ", length(pack("s!")), "\n";
print length(pack("l")), " ", length(pack("l!")), "\n";
```

i! et *I!* fonctionne aussi mais uniquement dans un souci de complétude puisqu'ils sont totalement identique à *i* et *I*.

La taille réelle (en octets) des short, int, long et long long natifs sur la plate-forme où Perl a été installé est aussi disponible via *Config* :

```
use Config;
print $Config{shortsize}, "\n";
print $Config{intsize}, "\n";
print $Config{longsize}, "\n";
print $Config{longlongsize}, "\n";
```

`{$Config{longlongsize}}` sera indéfini (undef) si votre système ne supporte pas les long long).



Les formats entiers *s*, *S*, *i*, *I*, *l*, *L*, *j* et *J* sont par construction non portables entre processeurs et entre systèmes d'exploitation puisqu'ils respectent l'ordre (big-endian ou little-endian) natif. Par exemple l'entier sur 4 octets 0x12345678 (305419896 en décimal) devrait être nativement ordonné (stocké et manipulé par les registres de la CPU) en octets comme :

```
0x12 0x34 0x56 0x78    # ordre little-endian
0x78 0x56 0x34 0x12    # ordre big-endian
```

perlfunc – Fonctions Perl prédéfinies

À la base, les CPU des familles Intel et VAX sont little-endian alors que tous les autres, par exemple Motorola m68k/88k, PPC, Sparc, HP PA, Power et Cray, sont big-endian. Les puces Alpha et MIPS peuvent être les deux : Digital les utilise(ait) en mode little-endian ; SGI/Cray les utilise en mode big-endian.

Les noms ``big-endian" et ``little-endian" sont des références au grand classique ``Les voyages de Gulliver" (au travers de l'article ``On Holy Wars and a Plea for Peace" par Danny Cohen, USC/ISI IEN 137, April 1, 1980) et donc aux habitudes des mangeurs d'oeufs Lilliputiens.

Quelques systèmes peuvent même avoir un ordre des octets encore plus bizarre tel que :

```
0x56 0x78 0x12 0x34
0x34 0x12 0x78 0x56
```

Vous pouvez voir les préférences de votre systèmes par

```
print join(" ", map { sprintf "%#02x", $_ }
            unpack("C*",pack("L",0x12345678))), "\n";
```

L'ordre des octets de la plate-forme où Perl a été installé est aussi disponible via *Config* :

```
use Config;
print $Config{byteorder}, "\n";
```

Les ordres d'octets '1234' et '12345678' sont little-endian alors que '4321' et '87654321' sont big-endian.

Si vous voulez des entiers compactés portables, il vous faut utiliser les formats `n`, `N`, `v` et `V` puisque leur taille et l'ordre de leurs octets sont connus. Voir aussi [the perlport manpage](#).

- ◆ Les nombres réels (simple et double précision) sont uniquement au format natif de la machine. À cause de la multiplicité des formats existants pour les flottants et du manque d'une représentation ``réseau" standard, aucune possibilité d'échange n'est proposée. Cela signifie que les données réelles compressées sur une machine peuvent ne pas être lisibles par une autre machine -- même si elles utilisent toutes deux l'arithmétique flottante IEEE (puisque l'ordre des octets de la représentation mémoire ne fait pas partie des spécifications IEEE). Voir aussi [the perlport manpage](#).

Sachez que Perl utilise des doubles en interne pour tous les calculs numériques et qu'une conversion de double vers float puis retour vers double amène à une perte de précision (i.e., `unpack("f", pack("f", $foo))` est généralement différent de `$foo`).

- ◆ Vous devez réaliser vous-même tous les alignements et les remplissages nécessaires en insérant par exemple assez de `x` lors du compactage. `pack()` et `unpack()` n'ont aucun moyen de savoir d'où viennent et où iront les octets. Par

perlfunc – Fonctions Perl prédéfinies

conséquent, `pack()` (et `unpack()`) gère leurs entrée et leurs sortie comme des séquences d'octets à plat.

- ◆ Un `()`-groupe est un sous-TEMPLATE entouré de parenthèses. Un tel groupe peut être répété soit en le postfixant par un compteur de répétition soit en le préfixant par un compteur via le caractère `/`.
- ◆ `x` et `X` accepte le modificateur `!`. Dans ce cas, ils agissent comme des commandes d'alignement : ils sautent vers la position la plus proche alignée sur un multiple de `count` octets. Par exemple, pour compacter ou décompacter la structure C `struct {char c; double d; char cc[2]}`, on peut utiliser le TEMPLATE `C x![d] d C[2]`; Cela garantit que les doubles seront alignés sur la taille d'un double.

Pour les commandes d'alignement un `count` de 0 est équivalent à un `count` de 1 ; Et tous les deux sont sans effet.
- ◆ Dans TEMPLATE, un commentaire commence par `#` et se termine en fin de ligne.
- ◆ Si TEMPLATE nécessite plus d'argument que ceux réellement fournis, `pack()` supposera que des arguments " " supplémentaires sont fournis. Si TEMPLATE nécessite moins d'arguments que ceux réellement fournis à `pack()`, les arguments en trop sont ignorés.

Exemples :

```
$foo = pack("CCCC", 65, 66, 67, 68);
# foo eq "ABCD"
$foo = pack("C4", 65, 66, 67, 68);
# même chose
$foo = pack("U4", 0x24b6, 0x24b7, 0x24b8, 0x24b9);
# même chose avec des lettres cerclées Unicode

$foo = pack("ccxxcc", 65, 66, 67, 68);
# foo eq "AB\0\0CD"

# note : les exemples précédents utilisant "C" et "c" ne sont
# corrects que sur des systèmes ACSII ou dérivés comme ISO Latin 1
# ou UTF-8. En EBCDIC, le premier exemple devrait être
# $foo = pack("CCCC", 193, 194, 195, 196);

$foo = pack("s2", 1, 2);
# "\1\0\2\0" sur little-endian
# "\0\1\0\2" sur big-endian

$foo = pack("a4", "abcd", "x", "y", "z");
# "abcd"

$foo = pack("aaaa", "abcd", "x", "y", "z");
# "axyz"
```

perlfunc – Fonctions Perl prédéfinies

```
$foo = pack("a14","abcdefg");
# "abcdefg\0\0\0\0\0\0\0"

$foo = pack("i9p1", gmtime);
# une vraie struct tm (sur mon système en tous cas)

$utmp_template = "Z8 Z8 Z16 L";
$utmp = pack($utmp_template, @utmp1);
# une struct utmp (BSD-isme)

@utmp2 = unpack($utmp_template, $utmp);
# "@utmp1" eq "@utmp2"

sub bintodec {
    unpack("N", pack("B32", substr("0" x 32 . shift, -32)));
}

$foo = pack('sx2l', 12, 34);
# short 12, two zero bytes padding, long 34
$bar = pack('s@4l', 12, 34);
# short 12, zero fill to position 4, long 34
# $foo eq $bar
```

La même valeur de TEMPLATE peut généralement être utilisée avec la fonction `unpack()`.

package

package NAMESPACE

Déclare l'unité de compilation comme faisant partie de l'espace de nommage NAMESPACE. La portée de la déclaration `package` (N.d.t : paquetage en français) commence à la déclaration elle-même et se termine à la fin du bloc, du fichier ou de l'eval englobant (c'est la même portée que l'opérateur `local()`). Tout identificateur dynamique non qualifié à venir sera dans cet espace de nommage. Une instruction `package` n'affecte que les variables dynamiques -- même celles sur lesquelles vous utilisez `local()` -- et *non* les variables lexicales créées par `my()`. C'est typiquement la première déclaration dans un fichier inclus par les opérateurs `require` ou `use`. Vous pouvez basculer vers un même paquetage en plusieurs endroits ; cela ne fait que changer la table de symboles utilisée par le compilateur pour la suite du bloc. Vous pouvez référencer des variables ou des descripteurs de fichiers d'autres packages en préfixant l'identificateur par le nom du package suivi de deux fois deux points : `$Package::Variable`. Si le nom de paquetage est vide, c'est la paquetage `main` qui est utilisé. Donc `$$:sail` est équivalent à `$main::sail` (et aussi à `$main'sail` qui peut encore se voir dans du vieux code)..

Si NAMESPACE est omis alors il n'y a pas de paquetage courant et tous les identificateurs doivent être soit complètement qualifiés soit lexicaux. Mais nous vous déconseillons fortement d'utiliser cette fonctionnalité. Son utilisation peut amener à des comportements inattendus et peut même faire planter certaines versions de Perl. Cette fonctionnalité est dépréciée et sera supprimée dans une version ultérieure.

Voir [Paquetages in the perlmod manpage](#) pour des plus amples informations à propos des paquetages (packages), des modules et des classes. Voir [the perlsub manpage](#) pour tous les problèmes de portée.

pipe READHANDLE,WRITEHANDLE

Ouvre une paire de tubes (pipe) exactement comme l'appel système correspondant.

Remarquez qu'en faisant une boucle de tubes entre plusieurs processus vous risquez un inter-blocage (deadlock) à moins d'y faire très attention. De plus, sachez que les tubes de

Perl utilisent des tampons d'entrée/sortie. Donc, selon les applications, vous aurez peut-être à positionner correctement `$|` pour vider vos WRITEHANDLE après chaque commande. Voir [the IPC::Open2 manpage](#), [the IPC::Open3 manpage](#) et [Communication Bidirectionnelle avec un autre Processus in the perlipc manpage](#) pour des exemples sur tout ça. Sur les systèmes qui supportent le drapeau fermeture-à-l-exécution (`close-on-exec`) sur les fichiers, ce drapeau sera positionné pour de nouveaux descripteurs de fichier en fonction de la valeur de `$^F`. Voir [\\$^F in the perlvar manpage](#).

pop TABLEAU

pop

Dépile et retourne la dernière valeur du tableau TABLEAU. La taille du tableau est diminué d'une unité. Cette commande a le même effet que :

```
$TABLEAU[ $#TABLEAU-- ]
```

Si le tableau est vide, c'est la valeur undef qui est retournée (cela peut arriver dans d'autres cas aussi). Si TABLEAU est omis, pop dépilera soit le tableau @ARGV dans le programme principale soit @_ dans les sous-routines, exactement comme [shift\(\)](#).

pos SCALAIRE

pos

Retourne l'offset (l'index du caractère dans la chaîne) où s'est arrêté la dernière recherche `m//g` sur la variable SCALAIRE spécifiée (s'appliquera à `$_` si la variable SCALAIRE n'est pas spécifiée). Peut être utilisé pour modifier cette offset. Une telle modification influe aussi sur l'assertion de longueur nulle `\G` dans les expressions rationnelles. Voir [the perlr](#) [manpage](#) et [the perlop manpage](#).

print DESCRIPTEUR LISTE

print LISTE

print

Affiche une chaîne ou une liste de chaînes. Retourne true (vrai) en cas de succès. DESCRIPTEUR peut être une variable scalaire auquel cas cette variable contiendra le nom ou la référence du DESCRIPTEUR et donc introduira un niveau d'indirection supplémentaire. (REMARQUE : si DESCRIPTEUR est une variable et que l'élément syntaxique qui suit est un terme, il peut être interprété accidentellement comme un opérateur à moins d'ajouter un + ou de mettre des parenthèses autour des arguments.) Si aucun DESCRIPTEUR n'est spécifié, affichera par défaut sur la sortie standard (ou le dernier canal de sortie sélectionné -- voir [select](#)). Si la liste LISTE est elle aussi omise, affichera `$_` sur le canal de sortie courant. Pour utiliser un autre canal que STDOUT comme canal de sortie par défaut, utilisez l'opération `select`. La valeur courante de `$,` (si elle existe) est affichée entre chaque item de LISTE. Remarquez qu'étant donné que `print` utilise une liste (LISTE), tout ce qui est dans LISTE est évalué dans un contexte de liste et, en particulier, toutes les expressions évaluées dans les sous-routines appelées le seront dans un contexte de liste. Faites aussi attention de ne pas faire suivre le mot-clé `print` par une parenthèse ouvrante à moins de vouloir clore la liste de ses arguments à la parenthèse fermante correspondante -- sinon préfixez votre parenthèse par un + ou entourez tous les arguments par des parenthèses.

Notez que si vous stockez vos DESCRIPTEUR dans un tableau (ou quelque chose qui nécessite une expression), vous devrez utiliser un bloc qui retourne une valeur :

```
print { $files[$i] } "stuff\n";
print { $OK ? STDOUT : STDERR } "stuff\n";
```

printf DESCRIPTEUR FORMAT, LISTE

***printf* FORMAT, LISTE**

Équivalent à `print DESCRIPTEUR sprintf(FORMAT, LISTE)` sauf que `$\` (le séparateur d'enregistrements en sortie) n'est pas ajouté. Le premier argument de la liste sera interprété comme le format de `printf()`. Voir `sprintf` pour les explications concernant le format. Si `use locale` est actif, le caractère utilisé comme séparateur décimal pour les nombres réels sera dépendant de la valeur de locale spécifiée dans `LC_NUMERIC`. Voir [the perllocale manpage](#).

Ne tombez pas de le piège d'utiliser `printf()` alors qu'un simple `print()` suffirait. `print()` est plus efficace et moins sujet à erreur.

***prototype* FONCTION**

Retourne, sous forme de chaîne, le prototype d'une fonction (ou `undef` si la fonction n'a pas de prototype). `FONCTION` est une référence ou le nom de la fonction dont on veut retrouver le prototype.

Si `FONCTION` est une chaîne commençant par `CORE::`, la suite de la chaîne se réfère au nom d'une fonction interne de Perl. Si la fonction interne n'est pas *redéfinissable* (par exemple `qw//`) ou si ses arguments ne peuvent s'exprimer sous forme de prototype (par exemple `system()`) – en d'autres termes, si la fonction interne ne se comporte pas comme une fonction Perl – la fonction prototype retournera `undef`. Sinon, c'est la chaîne qui décrit le prototype qui est retournée.

***push* TABLEAU,LISTE**

Traite `TABLEAU` comme une pile et empile les valeurs de la liste `LISTE` à la fin du tableau `TABLEAU`. La longueur de `TABLEAU` est augmentée de la longueur de la liste `LISTE`. Cela a le même effet que :

```
for $value (LISTE) {
    $TABLEAU[++$#TABLEAU] = $value;
}
```

mais en plus efficace. Retourne le nouveau nombre d'éléments du tableau.

q/CHAINE/

qq/CHAINE/

qr/CHAINE/

qx/CHAINE/

qw/CHAINE/

Guillemets/Apostrophes généralisées. Voir [the perlop manpage](#).

***quotemeta* EXPR**

quotemeta

Retourne la valeur de `EXPR` avec tous les caractères non alphanumériques précédés par un backslash (une barre oblique inverse). (Tous les caractères non reconnus par `/[A-Za-z_0-9]/` seront précédés d'un backslash quels que soient les réglages des locale). C'est la fonction interne qui implémente la séquence d'échappement `\Q` dans les chaînes entre guillemets.

Si `EXPR` est omis, s'appliquera à `$_`.

***rand* EXPR**

rand

Retourne un nombre fractionnaire aléatoire plus grand ou égal à 0 et plus petit que la valeur de `EXPR` (la valeur de `EXPR` devrait être positive). À défaut de `EXPR`, c'est 1 qui est utilisé comme borne. Actuellement, si `EXPR` vaut 0, c'est aussi un cas spécial qui est équivalent à

un appel avec 1 – ceci n'est documenté que depuis la version 5.8.0 de perl et peut changer dans des futures versions de perl. `rand()` appelle automatiquement `srand()` sauf cela a déjà été fait. Voir aussi `srand()`.

(Remarque : si votre fonction rand retourne régulièrement des nombres trop grands ou trop petits alors votre version de Perl a probablement été compilée avec une valeur erronée pour RANDBITS.)

read DESCRIPTEUR,SCALAIRE,LONGUEUR,OFFSET

read DESCRIPTEUR,SCALAIRE,LONGUEUR

Essaye de lire LONGUEUR caractères depuis le DESCRIPTEUR spécifié et les stocke dans la variable SCALAIRE. Retourne le nombre de caractères réellement lus, 0 à la fin du fichier ou undef si une erreur a lieu. La taille de la variable SCALAIRE augmentera ou diminuera pour atteindre la taille exacte de ce qui est lu. Un OFFSET (décalage) peut être spécifié pour placer les données lues ailleurs qu'au début de la chaîne. Cette fonction est implémentée par des appels à la fonction `fread()` de Perl ou du système. Pour obtenir un véritable appel système `read(2)`, voir `sysread()`.

Notez que l'on parle bien de caractères : selon l'état du DESCRIPTEUR, ce seront soit des octets (8-bit) soit des caractères qui seront lus. Par défaut tous les DESCRIPTEURS opèrent sur des octets mais, par exemple, si un DESCRIPTEUR a été ouvert avec le filtre d'entrée/sortie `:utf8` (voir `open` et la directive `open` dans `open`) alors les entrées/sorties se feront sur des caractères et non sur des octets.

readdir DIRHANDLE

Retourne l'entrée suivante d'un répertoire ouvert par `opendir()`. Dans un contexte de liste, retournera toutes les entrées restant dans le répertoire. Si il n'y a plus d'entrée, retournera la valeur undef dans un contexte scalaire ou une liste vide dans un contexte de liste.

Si vous prévoyez de faire des tests de fichiers sur les valeurs retournées par `readdir()`, n'oubliez pas de les préfixer par le répertoire en question. Sinon, puisqu'aucun appel à `chdir()` n'est effectué, vous risquez de tester un mauvais fichier.

```
opendir(DIR, $some_dir) || die "can't opendir $some_dir: $!";
@dots = grep { /^\.\/ && -f "$some_dir/$_" } readdir(DIR);
closedir DIR;
```

readline EXPR

Lit à partir du descripteur dont le typeglob est donné par EXPR. Dans un contexte scalaire, chaque appel lit et retourne la ligne suivante jusqu'à atteindre la fin du fichier. À ce moment, un appel supplémentaire retournera undef. Dans un contexte de liste, la lecture se fera jusqu'à la fin du fichier et le résultat sera une liste de lignes. La notion de "ligne" utilisée ici est celle définie par la variable `$/` ou `$INPUT_RECORD_SEPARATOR`. Voir `$/ in the perlvar manpage`.

Lorsque `$/` est positionné à `undef`, que `readline()` est utilisé dans un contexte scalaire (i.e. en mode 'slurp') et que le fichier lu est vide, le premier appel retourne '' et `undef` pour les suivants.

C'est la fonction interne qui implémente l'opérateur `<EXPR>` mais vous pouvez l'utiliser directement. L'opérateur `<EXPR>` est décrit en détail dans `Les opérateurs d'E/S in the perlop manpage`.

```
$line = <STDIN>;
$line = readline(*STDIN);           # même chose
```

readlink EXPR

readlink

Retourne la valeur d'un lien symbolique si les liens symboliques sont implémentés. Sinon, produit une erreur fatale. Si il y a une erreur système, cette fonction retournera la valeur undef et positionnera la variable \$! (errno). Si EXPR est omis, s'applique à \$_.

readpipe EXPR

EXPR est exécuté comme une commande système. La sortie standard de la commande est collectée puis retournée. Dans un contexte scalaire, cette sortie est une seule chaîne (contenant éventuellement plusieurs lignes). Dans un contexte de liste, retourne une liste de lignes (telles que définies par la variable \$/ ou \$INPUT_RECORD_SEPARATOR). C'est la fonction interne qui implémente l'opérateur `qx/EXPR/` mais vous pouvez l'utiliser directement. L'opérateur `qx/EXPR/` est décrit plus en détail dans [Les opérateurs d'E/S in the perlop manpage](#).

recv SOCKET,SCALAIRE,LONGUEUR,FLAGS

Reçoit un message depuis un socket. Tente de recevoir LONGUEUR caractères de données dans la variable SCALAIRE depuis le descripteur spécifié par SOCKET. SCALAIRE grossira ou réduira jusqu'à la taille des données réellement lues. Utilise les mêmes FLAGS que l'appel système du même nom. Retourne l'adresse de l'émetteur si le protocole de SOCKET le permet ; retourne une chaîne vide sinon. Retourne la valeur undef en cas d'erreur. Cette fonction est implémentée en terme d'appel système `recvfrom(2)`. Voir [UDP : Transfert de Message in the perlipc manpage](#) pour des exemples. Notez que l'on parle bien de *caractères* : selon l'état du SOCKET, ce seront soit des octets (8-bit) soit des caractères qui seront lus. Par défaut tous les SOCKETs opèrent sur des octets mais si un SOCKET a été modifié via `binmode()` pour utiliser le filtre d'entrée/sortie `:utf8` (voir la directive `open` dans `open`) alors les entrées/sorties se feront sur des caractères et non sur des octets.

redo LABEL

redo

La commande `redo` redémarre une boucle sans évaluer à nouveau la condition. Le bloc `continue`, s'il existe, n'est pas évalué. Si l'étiquette LABEL est omise, la commande se réfère à la boucle englobante la plus profonde. Cette commande est habituellement utilisée par des programmes qui veulent utiliser eux-mêmes ce qui vient juste d'être lu :

```
# un programme simple pour supprimer les commentaires en Pascal
# (Attention: suppose qu'il n'y pas de { ni de } dans les chaînes.)
LINE: while (<STDIN>) {
    while (s|({.*}.*){.*}|$1 |) {}
    s|{.*}| |;
    if (s|{.*}| |) {
        $front = $_;
        while (<STDIN>) {
            if (/)/) { # end of comment?
                s|^|$front\{|;
                redo LINE;
            }
        }
    }
}
print;
```

`redo` ne peut pas être utilisé pour redémarrer un bloc qui doit retourner une valeur comme `eval {}`, `sub {}` ou `do {}` et ne devrait pas être utilisé pour sortir d'une opération `grep()` ou `map()`.

Remarques qu'un bloc en lui-même est sémantiquement identique à une boucle qui ne

s'exécute qu'une fois. Donc `redo` dans un tel bloc le transforme effectivement en une construction de boucle.

Voir aussi `continue` pour illustrer la manière dont `last`, `next` et `redo` fonctionnent.

ref *EXPR*

ref

Retourne une valeur true (vraie) si *EXPR* est une référence, false (faux) sinon. Si *EXPR* n'est pas spécifié, s'applique à `$_`. La valeur retournée dépend du type de ce qui est référencé par la référence. Les types internes incluent :

```
SCALAR
ARRAY
HASH
CODE
REF
GLOB
LVALUE
```

Si l'objet référencé a été béni (par `bless()`) par un paquetage alors le nom du paquetage est retourné. Vous pouvez voir `ref()` comme une sorte d'opérateur `typeof()` (type de).

```
if (ref($r) eq "HASH") {
    print "r is a reference to a hash.\n";
}
unless (ref($r)) {
    print "r is not a reference at all.\n";
}
if (UNIVERSAL::isa($r, "HASH")) { # for subclassing
    print "r is a reference to something that isa hash.\n";
}
```

Voir aussi [the perlref manpage](#).

rename *ANCIENNOM,NOUVEAUNOM*

Change le nom d'un fichier ; un fichier préexistant de nom *NOUVEAUNOM* sera écrasé. Retourne true (vrai) en cas de succès ou false (faux) sinon.

Le comportement de cette fonction varie beaucoup d'un système à l'autre. Par exemple, habituellement elle ne fonctionne pas entre des systèmes de fichiers différents même si la commande système `mv` réussit parfois à le faire. Il y a d'autres restrictions selon que cela fonctionne ou non sur des répertoires, sur des fichiers ouverts ou sur des fichiers préexistants. Regarder [the perlport manpage](#) et la page de documentation de `rename(2)` pour les détails.

require *VERSION*

require *EXPR*

require

Exige une version de Perl spécifiée par *VERSION* ou une « sémantique » spécifiée par *EXPR* ou par `$_` si *EXPR* n'est pas fourni.

VERSION peut être spécifié soit sous la forme d'une valeur numérique telle que 5.006, elle sera alors comparée à `$]` soit sous la forme d'une valeur littérale telle que v5.6.1 qui sera alors comparée à `$^V` (ou son synonyme `$PERL_VERSION`). Une erreur fatale est produite lors de l'exécution si *VERSION* est supérieure à la version de l'interpréteur Perl courant.

Comparez avec `use` qui peut faire un contrôle similaire mais lors de la compilation.

```
require v5.6.1;      # contrôle de version à l'exécution
require 5.6.1;      # idem
require 5.006_001;  # argument numérique autorisé par compatibilité
```

Sinon, exige que le fichier d'une bibliothèque soit inclus si ce n'est pas déjà fait. Le fichier est inclus via le mécanisme `do-FICHER` qui est pratiquement une variante de `eval()`. Sa

sémantique est similaire à la procédure suivante :

```
sub require {
    my($filename) = @_ ;
    return 1 if $INC{$filename};
    my($realfilename,$result);
    ITER: {
        foreach $prefix (@INC) {
            $realfilename = "$prefix/$filename";
            if (-f $realfilename) {
                $INC{$filename} = $realfilename;
                $result = do $realfilename;
                last ITER;
            }
        }
        die "Can't find $filename in \@INC";
    }
    delete $INC{$filename} if $@ || !$result;
    die $@ if $@;
    die "$filename did not return true value" unless $result;
    return $result;
}
```

Remarquez que le fichier ne sera pas inclus deux fois sous le même nom. Le fichier doit retourner true (vrai) par sa dernière instruction pour indiquer une exécution correcte du code d'initialisation. Il est donc courant de terminer un tel fichier par un ``1;" à moins d'être sûr qu'il retournera true (vrai) par un autre moyen. Mais il est plus sûr de mettre ``1;" au cas où vous ajouteriez quelques instructions.

Si `EXPR` est un nom simple (bareword), `require` suppose l'extension ``.pm" et remplace pour vous les ``:" par des ``/' dans le nom du fichier afin de rendre plus simple le chargement des modules standards. Cette forme de chargement des modules ne risque pas d'altérer votre espace de noms.

En d'autres termes, si vous dites :

```
require Foo::Bar; # un splendide mot simple
```

La fonction `require` cherchera en fait le fichier ``*Foo/Bar.pm*" dans les répertoires spécifiés par le tableau `@INC`.

Mais si vous essayez :

```
$class = 'Foo::Bar';
require $class; # $class n'est pas un mot simple
#ou
require "Foo::Bar"; # n'est pas un mot simple à cause des guillemets
```

La fonction `require` cherchera le fichier ``*Foo::Bar*" dans les répertoires du tableau `@INC` et se plaindra qu'elle ne peut trouver le fichier ``*Foo::Bar*". Dans ce cas, vous pouvez faire :

```
eval "require $class";
```

Vous pouvez aussi accrocher des actions (on dit crochets ou des ``hooks") à la fonctionnalité d'importation en plaçant directement du code Perl dans le tableau `@INC`. Il y a trois formes de crochets : une référence vers une subroutine, une référence vers un tableau et une référence vers un objet bénit.

La référence vers une subroutine est le cas simple. Lorsque le système d'inclusion rencontre une telle référence en parcourant le tableau `@INC`, la subroutine est appelée avec deux arguments. Le premier est une référence vers la subroutine elle-même et le second est le nom du fichier à inclure (par exemple ``*Foo/Bar.pm*"). La subroutine devrait alors retourner soit `undef` soit un descripteur à partir duquel le fichier à inclure sera lu. Si la valeur de retour est `undef`, `require` continuera sa recherche dans la suite du tableau `@INC`. Si c'est une référence vers un tableau, le premier élément de ce tableau doit être une référence vers une subroutine. Cette subroutine sera appelée comme ci-dessous mais le premier argument sera la référence vers le tableau. Cela permet de passer des arguments

indirectement à la subroutine.

En d'autres termes, vous pouvez écrire :

```
push @INC, \&my_sub;
sub my_sub {
    my ($coderef, $filename) = @_; # $coderef est \&my_sub
    ...
}
```

ou :

```
push @INC, [ \&my_sub, $x, $y, ... ];
sub my_sub {
    my ($arrayref, $filename) = @_;
    # On retrouve $x, $y, ...
    my @parameters = @$arrayref[1..$#$arrayref];
    ...
}
```

Si c'est une référence vers un objet, cet objet doit avoir une méthode INC qui sera appelée comme ci-dessus avec comme premier paramètre l'objet lui-même. (Notez que vous devez complètement qualifier le nom de cette subroutine pour éviter qu'il soit placé de force dans le package C<main>.) Voici un exemple typique de S<code >:

```
# Dans Foo.pm
package Foo;
sub new { ... }
sub Foo::INC {
    my ($self, $filename) = @_;
    ...
}

# Dans le programme principal (main)
push @INC, new Foo(...);
```

Notez que ces crochets ont aussi le droit de remplir les données dans %INC correspondant aux fichiers qu'ils ont chargés. Voir [%INC in the perlvar manpage](#).

Pour une fonctionnalité d'importation encore plus puissante, voir [use](#) et [the perlmod manpage](#).

reset EXPR

reset

Généralement utilisée dans un bloc [continue](#) à la fin d'une boucle pour effacer les variables et réinitialiser les recherches ?? pour qu'elle marche à nouveau. L'expression EXPR est interprétée comme une liste de caractères (le moins est autorisé pour des intervalles). Toutes les variables commençant par l'un de ces caractères sont réinitialisées à leur état primitif. Si EXPR est omis, les motifs de recherche qui ne marchent qu'une fois (?motif?) sont réinitialisés pour fonctionner à nouveau. Ne réinitialise que les variables et les motifs du paquetage courant. Retourne toujours 1. Exemples :

```
reset 'X';           # réinitialise toutes les variables X...
reset 'a-z';        # réinitialise toutes les variables
                    # commençant par une minuscule
reset;              # réinitialise juste les motifs ?...?
```

Réinitialiser "A-Z" n'est pas recommandé parce que cela efface les tableaux @ARGV et @INC ainsi que la table de hachage %ENV. Ne réinitialise que les variables de paquetage — les variables lexicales ne sont pas modifiées mais elles s'effacent toutes seules dès que l'on sort de leur portée, ce qui devrait vous inciter à les utiliser. Voir [my](#).

return EXPR

return

Sort d'une subroutine, d'un bloc `eval()` ou d'un `do FICHIER` en retournant la valeur donnée par `EXPR`. L'évaluation de `EXPR` peut se faire dans un contexte scalaire, de liste ou vide selon la manière dont la valeur sera utilisée. Le contexte peut varier d'une exécution à l'autre (voir `wantarray()`). Si aucune `EXPR` n'est donnée, retourne la liste vide dans un contexte de liste, la valeur `undef` dans un contexte scalaire et rien du tout dans un contexte vide.

(Remarque : en l'absence de `return` explicite, une subroutine, un bloc `eval` ou un `do FICHIER` retournera automatiquement la valeur de la dernière expression évaluée.)

reverse LISTE

Dans un contexte de liste, retourne une liste de valeurs constituée des éléments de `LISTE` en ordre inverse. Dans un contexte scalaire, concatène les éléments de `LISTE` et retourne la chaîne ainsi constituée mais avec les caractères dans l'ordre inverse.

```
print reverse <>;           # tac (cat à l'envers) les lignes,
                           # la dernière ligne en premier
undef $/;                  # pour un <> efficace
print scalar reverse <>;   # tac (cat à l'envers) les octets,
                           # la dernière ligne en reimerp
```

Cet opérateur est aussi utilisé pour inverser des tables de hachage bien que cela pose quelques problèmes. Si une valeur est dupliquée dans la table originale, seule l'une des ces valeurs sera représentée comme une clé dans la table résultante. Cela nécessite aussi de mettre toute la table à plat avant d'en reconstruire une nouvelle ce qui peut prendre beaucoup de temps sur une grosse table telle qu'un fichier DBM.

```
%by_name = reverse %by_address;    # Inverse la table
```

rewinddir DIRHANDLE

Ramène la position courante au début du répertoire pour le prochain `readdir()` sur `DIRHANDLE`.

rindex CHAINE,SUBSTR,POSITION

rindex CHAINE,SUBSTR

Fonctionne exactement comme `index` sauf qu'il retourne la position de la dernière occurrence de `SUBSTR` dans `CHAINE`. Si `POSITION` est spécifiée, retourne la dernière occurrence avant ou exactement à cette position.

rmdir REPNOM

rmdir

Efface le répertoire spécifié par `REPNOM` si ce répertoire est vide. En cas de succès, retourne `true` (vrai) ou sinon, retourne `false` (faux) et positionne la variable `$!` (`errno`). Si `REPNOM` est omis, utilise `$_`.

s//

L'opérateur de substitution. Voir [the perlop manpage](#).

scalar EXPR

Contraint l'interprétation de `EXPR` dans un contexte scalaire et retourne la valeur de `EXPR`.

```
@counts = ( scalar @a, scalar @b, scalar @c );
```

Il n'y pas d'opérateur équivalent pour contraindre l'interprétation d'une expression dans un contexte de liste parce qu'en pratique ce n'est jamais nécessaire. Si vous en avez réellement besoin, vous pouvez utiliser une construction comme `@{ [(une expression)] }` mais un simple `(une expression)` suffit en général.

Comme `scalar` est un opérateur unaire, si vous l'utilisez accidentellement sur une EXPR constituée d'une liste entre parenthèses, cela se comporte comme l'opérateur scalaire virgule en évaluant tous les éléments dans un contexte vide sauf le dernier élément qui est évalué dans un contexte scalaire et retourné. C'est rarement ce que vous vouliez.

L'instruction suivante :

```
print uc(scalar(&foo,$bar)), $baz;
```

est équivalente à ces deux instructions :

```
&foo;
print(uc($bar), $baz);
```

Voir [the perlop manpage](#) pour les détails sur les opérateurs unaires et l'opérateur virgule.

seek *DESCRIPTEUR, POSITION, WHENCE*

Modifie la position d'un DESCRIPTEUR exactement comme le fait l'appel `fseek()` de `stdio()`. DESCRIPTEUR peut être une expression dont la valeur donne le nom du descripteur. Les valeurs possibles de WHENCE sont 0 pour régler la nouvelle position *en octets* à POSITION, 1 pour la régler à la position courante plus POSITION ou 2 pour la régler à EOF plus POSITION (en général négative). Pour WHENCE, vous pouvez utiliser les constantes `SEEK_SET`, `SEEK_CUR` et `SEEK_END` (relatif au début du fichier, à la position courante, à la fin du fichier) provenant du module `Fcntl`. Renvoie 1 en cas de succès et 0 sinon.

Notez bien qu'on parle *en octets* : même si le DESCRIPTEUR a été configuré pour opérer sur des caractères (par exemple en utilisant le filtre `:utf8`), `seek()` travaille quand même en terme d'octets et non en terme de caractères (implémenter cette fonctionnalité aurait rendu `seek()` et `tell()` extrêmement lents).

Si vous voulez régler la position pour un fichier dans le but d'utiliser `sysread()` ou `syswrite()`, n'utilisez pas `seek()` -- la bufferisation rend ses effets imprévisibles et non portables. Utilisez `sysseek()` à la place.

À cause des règles et de la rigueur du C ANSI, sur certains systèmes, vous devez faire un `seek` à chaque fois que vous basculez entre lecture et écriture. Entre autres choses, cela a pour effet d'appeler la fonction `clearerr(3)` de `stdio`. Un WHENCE de 1 (`SEEK_CUR`) est pratique pour ne pas modifier la position dans le fichier :

```
seek(TEST,0,1);
```

C'est aussi très pratique pour les applications qui veulent simuler `tail -f`. Une fois rencontré un EOF en lecture et après avoir attendu un petit peu, vous devez utiliser `seek()` pour réactiver les choses. L'appel à `seek()` ne modifie pas la position courante mais par contre, il efface la condition fin-de-fichier (EOF) sur le descripteur et donc, au prochain `<FILE>`, Perl essaiera à nouveau de lire quelque chose. Espérons-le.

Si cela ne marche pas (certaines implémentations des E/S sont particulièrement irascibles) alors vous devrez faire quelque chose comme :

```
for (;;) {
    for ($curpos = tell(FILE); $_ = <FILE>;
        $curpos = tell(FILE)) {
        # search for some stuff and put it into files
    }
    sleep($for_a_while);
    seek(FILE, $curpos, 0);
}
```

seekdir *DIRHANDLE, POS*

Règle la position courante pour la routine `readdir()` sur un DIRHANDLE. POS doit être une valeur retournée par `tellidir()`. Possède les mêmes limitations que l'appel système correspondant.

select DESCRIPTEUR**select**

Retourne le descripteur courant. Sélectionne le descripteur DESCRIPTEUR comme sortie par défaut si DESCRIPTEUR est fourni. Ceci a deux effets : tout d'abord, un `write()` ou un `print()` sans descripteur spécifié iront par défaut sur ce DESCRIPTEUR. Ensuite, toutes références à des variables relatives aux sorties se référeront à ce canal de sortie. Par exemple, si vous devez spécifier un en-tête de format pour plusieurs canaux de sortie, vous devez faire la chose suivante :

```
select(REPORT1);
$^ = 'report1_top';
select(REPORT2);
$^ = 'report2_top';
```

DESCRIPTEUR peut être une expression dont le résultat donne le nom du descripteur réel. Donc :

```
$oldfh = select(STDERR); $| = 1; select($oldfh);
```

Certains programmeurs préfèrent considérer les descripteurs comme des objets avec des méthodes. Ils écriraient donc l'exemple précédent de la manière suivante :

```
use IO::Handle;
STDERR->autoflush(1);
```

select RBITS, WBITS, EBITS, TIMEOUT

Ceci utilise directement l'appel système `select(2)` avec les masques de bit spécifiés qui peuvent être construits en utilisant `fileno()` et `vec()` comme dans les lignes suivantes :

```
$rin = $win = $sein = '';
vec($rin, fileno(STDIN), 1) = 1;
vec($win, fileno(STDOUT), 1) = 1;
$sein = $rin | $win;
```

Si vous voulez surveiller de nombreux descripteurs, vous aurez peut-être à écrire une subroutine :

```
sub fhbits {
    my(@fhlist) = split(' ', $_[0]);
    my($bits);
    for (@fhlist) {
        vec($bits, fileno($_), 1) = 1;
    }
    $bits;
}
$rin = fhbits('STDIN TTY SOCK');
```

L'appel classique est :

```
($nfound, $timeleft) =
    select($rout=$rin, $wout=$win, $eout=$sein, $timeout);
```

ou pour attendre que quelque chose soit prêt :

```
$nfound = select($rout=$rin, $wout=$win, $eout=$sein, undef);
```

De nombreux systèmes ne prennent pas la peine de retourner quelque chose d'utile dans `$timeleft` (le temps restant). En conséquence, un appel à `select()` dans un contexte scalaire retourne juste `$nfound`.

`undef` est une valeur acceptable pour les masques de bits. Le timeout, s'il est spécifié, est donné en secondes et peut être fractionnaire. Note : certaines implémentations ne sont pas capables de retourner `$timeleft`. Dans ce cas, elles retournent toujours un `$timeleft` égal au `$timeout` fourni.

Vous pouvez spécifier une attente de 250 millisecondes de la manière suivante :

```
select(undef, undef, undef, 0.25);
```

Notez que selon les implémentations, un `select` continuera ou non après réception d'un signal (un `SIGALRM` par exemple).

ATTENTION : il ne faut pas mélanger des E/S bufferisées (comme `read()` ou `<FH>`) avec `select()` excepté lorsque la norme POSIX le permet et, dans ce cas, uniquement sur les systèmes POSIX. Vous devez utiliser `sysread()` à la place.

semctl ID,SEMNUM,CMD,ARG

Appelle la fonction `semctl()` des IPC System V. Vous aurez sans doute besoin de :

```
use IPC::SysV;
```

au préalable pour avoir les définitions correctes des constantes. Si `CMD` est `IPC_STAT` ou `GETALL` alors `ARG` doit être une variable capable de contenir la structure `semid_ds` retournée ou le tableau des valeurs des sémaphores. Les valeurs retournées sont comme celles de `ioctl()` : la valeur `undef` en cas d'erreur, la chaîne `"0 but true"` pour rien ou la vraie valeur retournée dans les autres cas. `ARG` doit être un vecteur d'entiers courts (short) natifs qui peut être créé par `pack("s!", (0)x$sem)`. Voir aussi la documentation de `IPC::SysV` et de `IPC::SysV::Semaphore`.

semget KEY,NSEMS,FLAGS

Appelle la fonction `semget()` des IPC System V. Renvoie l'id du sémaphore ou la valeur `undef` en cas d'erreur. Voir aussi la documentation de `IPC::SysV` et de `IPC::SysV::Semaphore`.

semop KEY,OPSTRING

Appelle la fonction `semop()` des IPC System V pour réaliser certaines opérations sur les sémaphores comme l'attente ou la signalisation. `OPSTRING` doit être un tableau compacté (par `pack()`) de structures `semop`. Chaque structure `semop` peut être générée par `pack("s!3", $semnum, $semop, $semflag)`. Le nombre total d'opérations sur les sémaphores est déduit de la taille de `OPSTRING`. Renvoie `true` (vrai) en cas de succès ou `false` (faux) en cas d'erreur. Par exemple, le code suivant attend sur le sémaphore `$semnum` de l'ensemble de sémaphores d'id `$semid` :

```
$semop = pack("s!3", $semnum, -1, 0);
die "Semaphore trouble: $!\n" unless semop($semid, $semop);
```

Pour envoyer un signal au sémaphore, remplacez le `-1` par `1`. Voir aussi la documentation de `IPC::SysV` et de `IPC::SysV::Semaphore`.

send SOCKET,MSG,FLAGS,TO

send SOCKET,MSG,FLAGS

Envoie un message sur un socket. Envoie le scalaire `MSG` vers le descripteur `SOCKET`. Utilise les mêmes flags que l'appel système du même nom. Pour des sockets non connectés, vous devez spécifier dans le paramètre `TO` une destination pour l'envoi. Dans ce cas c'est la fonction C `sendto()` qui est utilisée. Retourne le nombre de caractères envoyés ou la valeur `undef` s'il y a une erreur. L'appel système `sendmsg(2)` n'est pas implémenté pour l'instant. Voir les exemples dans [UDP : Transfert de Message in the perlipc manpage](#).

Notez que l'on parle bien de *caractères* : selon l'état du `SOCKET`, ce seront soit des octets (8-bit) soit des caractères qui seront lus. Par défaut tous les `SOCKET`s opèrent sur des octets mais si un `SOCKET` a été modifié via `binmode()` pour utiliser le filtre d'entrée/sortie `:utf8` (voir la directive `open` dans `open`) alors les entrées/sorties se feront sur des caractères et non sur des octets.

setpgrp PID,PGRP

Modifie le groupe de processus courant pour le `PID` spécifié (0 pour le processus courant). Cela produira une erreur fatale si vous l'utilisez sur un système qui n'implémente pas l'appel

POSIX `setpgrp(2)` ou l'appel BSD `setpgrp(2)`. Si les arguments sont omis, ils prennent comme valeur par défaut 0, 0. Remarquez aussi que la version BSD 4.2 de `setpgrp()` n'accepte aucun argument. Donc seul `setpgrp 0, 0` est portable. Voir aussi `POSIX::setsid()`.

setpriority WHICH,WHO,PRIORITY

Modifie la priorité courante d'un process, d'un groupe de process ou d'un utilisateur. (Voir [setpriority\(2\)](#).) Cela produira une erreur fatale si vous l'utilisez sur un système qui n'implémente pas `setpriority(2)`.

setsockopt SOCKET,LEVEL,OPTNAME,OPTVAL

Modifie l'option spécifiée d'un socket. Retourne undef en cas d'erreur. OPTVAL peut être [undef](#) si vous ne voulez pas passer d'argument.

shift TABLEAU

shift

Retourne la première valeur d'un tableau après l'avoir supprimée du tableau en rétrécissant sa taille de 1 et en déplaçant tout vers le bas. Renvoie undef si il n'y a pas d'éléments dans le tableau. Si TABLEAU est omis, shift agira soit sur le tableau @_ s'il est dans la portée lexicale d'une subroutine ou d'un format, soit sur le tableau @ARGV s'il est dans la portée lexicale d'un fichier ou s'il est dans une portée lexicale établie par l'une des constructions `eval ''`, `BEGIN {}`, `END {}` ou `INIT {}`.

Voir aussi [unshift](#), [push](#) et [pop](#). [shift\(\)](#) et [unshift\(\)](#) agissent sur le côté gauche d'un tableau exactement comme [pop\(\)](#) et [push\(\)](#) le font sur le côté droit.

shmctl ID,CMD,ARG

Appelle la fonction shmctl de IPC System V. Vous aurez sans doute besoin de :

```
use IPC::SysV;
```

au préalable pour avoir les définitions correctes des constantes. Si CMD est `IPC_STAT` alors ARG doit être une variable capable de contenir la structure `shmids` retournée. Les valeurs retournées sont comme celles de `ioctl()` : la valeur undef en cas d'erreur, la chaîne `"0 but true"` pour zéro ou la vraie valeur retournée dans les autres cas. Voir aussi la documentation de `IPC::SysV`.

shmget KEY,SIZE,FLAGS

Appelle la fonction shmget de IPC System V. Renvoie l'id du segment de mémoire partagée ou undef en cas d'erreur. Voir aussi la documentation de `IPC::SysV`.

shmread ID,VAR,POS,SIZE

shmwrite ID,CHAINE,POS,SIZE

Lit ou écrit le segment de mémoire partagée System V d'id ID en commençant à la position POS et sur une taille de SIZE en s'attachant à lui puis en le lisant ou en l'écrivant puis enfin en s'en détachant. Lors d'une lecture, VAR doit être une variable qui contiendra les données lues. Lors d'une écriture, si CHAINE est trop long, seuls SIZE octets seront utilisés, si CHAINE est trop court des caractères nuls seront ajoutés pour compléter jusqu'à SIZE octets. Retourne true (vrai) en cas de succès et false (faux) en cas d'erreur. [shmread\(\)](#) souille (taint) la variable CHAINE. Voir aussi la documentation de `IPC::SysV`.

shutdown SOCKET,HOW

Ferme une connexion socket de la manière indiquée par HOW qui est interprété comme le fait l'appel système du même nom.

perlfunc – Fonctions Perl prédéfinies

```
shutdown(SOCKET, 0); # J'ai arrêté de lire des données
shutdown(SOCKET, 1); # J'ai arrêté d'écrire des données
shutdown(SOCKET, 2); # J'ai arrêté d'utiliser ce socket
```

C'est pratique pour des sockets pour lesquels vous voulez indiquer à l'autre extrémité que vous avez fini d'écrire mais pas de lire ou vice versa. C'est aussi une forme plus insistante de close puisque qu'elle désactive aussi le descripteur de fichier pour tous les process dupliqués par fork.

sin* *EXPR

sin

Retourne le sinus de EXPR (exprimé en radians). Si EXPR est omis, retourne le sinus de \$_.

Pour calculer la fonction inverse du sinus, vous pouvez utiliser la fonction

Math::Trig::asin ou utiliser cette relation :

```
sub asin { atan2($_[0], sqrt(1 - $_[0] * $_[0])) }
```

sleep* *EXPR

sleep

Demande au script de s'endormir pendant EXPR secondes ou pour toujours si EXPR est omis. Peut être interrompu si le process reçoit un signal comme SIGALRM. Renvoie la durée réelle du sommeil en secondes. Vous ne pourrez probablement pas mélanger des appels [alarm\(\)](#) et [sleep\(\)](#) car [sleep\(\)](#) est souvent implémenté en utilisant [alarm\(\)](#).

Sur quelques vieux systèmes, il se peut que la durée du sommeil soit d'une seconde de moins que celle que vous avez demandée en fonction de la manière dont il compte les secondes. Les systèmes plus modernes s'endorment toujours pendant la bonne durée. En revanche, il peut arriver que votre sommeil dure plus longtemps que prévu sur un système multi-tâches très chargé.

Pour des délais d'une granularité inférieure à la seconde, vous pouvez utiliser l'interface Perl [syscall\(\)](#) pour accéder à `setitimer(2)` si votre système le supporte ou sinon regarder [select\(\)](#) plus haut. Le module Time::HiRes (disponible sur CPAN ou intégré à la distribution standard depuis Perl 5.8.0) peut aussi aider.

Regarder aussi la fonction `sigpause()` du module POSIX.

socket* *SOCKET,DOMAIN,TYPE,PROTOCOL

Ouvre un socket du type spécifié et l'attache au descripteur SOCKET. DOMAIN, TYPE et PROTOCOL sont spécifiés comme pour l'appel système du même nom. Vous devriez mettre `use Socket;` au préalable pour importer les définitions correctes. Voir les exemples dans [Sockets : Communication Client/Serveur in the perlipc manpage](#).

Sur les systèmes qui supportent le drapeau fermeture-à-l-exécution (close-on-exec) sur les fichiers, ce drapeau sera positionné pour de nouveaux descripteurs de fichier en fonction de la valeur de \$^F. Voir [\\$^F in the perlvar manpage](#).

socketpair* *SOCKET1,SOCKET2,DOMAIN,TYPE,PROTOCOL

Crée une paire de sockets sans nom dans le domaine spécifié et du type spécifié. DOMAIN, TYPE et PROTOCOL sont spécifiés comme pour l'appel système du même nom. Si l'appel système n'est pas implémenté, cela produit une erreur fatale. Retourne true (vrai) en cas de succès.

Sur les systèmes qui supportent le drapeau fermeture-à-l-exécution (close-on-exec) sur les fichiers, ce drapeau sera positionné pour de nouveaux descripteurs de fichier en fonction de la valeur de \$^F. Voir [\\$^F in the perlvar manpage](#).

Certains systèmes définissent [pipe\(\)](#) en terme de [socketpair\(\)](#), auquel cas un appel à

pipe(Rdr, Wtr) est quasiment équivalent à :

```
use Socket;
socketpair(Rdr, Wtr, AF_UNIX, SOCK_STREAM, PF_UNSPEC);
shutdown(Rdr, 1);      # plus d'écriture pour le lecteur
shutdown(Wtr, 0);     # plus de lecture pour l'écrivain
```

Voir [the perlipc manpage](#) pour des exemples d'utilisation de socketpair. Perl 5.8 et plus émuleront socketpair en utilisant de sockets IP sur la machine locale si votre système implémente les sockets mais pas socketpair.

sort SUBNAME LISTE

sort BLOC LISTE

sort LISTE

Dans un contexte de liste, sort() trie LISTE et retourne la liste de valeurs triée. Dans un contexte scalaire, le comportement de sort() est indéfini.

Si SUBNAME et BLOC sont omis, le tri est effectué dans l'ordre standard de comparaison de chaînes. Si SUBNAME est spécifié, il donne le nom d'une subroutine qui retourne un entier plus petit, égal ou plus grand que 0 selon l'ordre dans lequel les éléments du tableau doivent être triés. (Les opérateurs `<=>` et `cmp` sont extrêmement utiles dans de telles subroutines.) SUBNAME peut être une variable scalaire, auquel cas sa valeur donne le nom de (ou la référence vers) la subroutine à utiliser. À la place de SUBNAME, vous pouvez fournir un BLOC comme subroutine de tri anonyme et en ligne.

Si le prototype de la subroutine est (`$$`), les éléments à comparer sont passés par référence dans `@_` comme pour une subroutine normale. C'est plus lent qu'une subroutine sans prototype pour laquelle les éléments à comparer sont passés à la subroutine par les variables globales du package courant `$a` et `$b` (voir exemples ci-dessous). Dans ce dernier cas, il est contre-productif de déclarer `$a` et `$b` comme des variables lexicales.

Dans tous les cas, la subroutine ne peut pas être récursive. Les valeurs à comparer sont toujours passées par référence. Donc, ne les modifiez pas.

Vous ne pouvez pas non plus sortir du bloc `sort` ou de la subroutine en utilisant un goto() ou les opérateurs de contrôle de boucles décrits dans [the perlsyn manpage](#).

Lorsque `use locale` est actif, sort LISTE trie LISTE selon l'ordre (collation) du locale courant. Voir [the perllocale manpage](#).

Jusqu'à Perl 5.6, Perl utilisait un algorithme quicksort pour implémenter le tri. Cet algorithme n'était pas stable et *pouvait* parfois être quadratique. (Un tri *stable* préserve l'ordre des éléments égaux. De plus, bien que la complexité moyenne d'un quicksort soit en $O(N \log N)$ pour un tableau de taille N , la complexité peut atteindre $O(N^2)$, un comportement quadratique, pour certains tableaux.) En Perl 5.7, l'implémentation du quicksort a été remplacée par un algorithme stable mergesort dont la complexité est $O(N \log N)$. Mais des tests de performance ont montré que, dans certains cas et sur certaines plateformes, quicksort était plus rapide. Perl 5.8 propose donc une directive `sort` afin de choisir l'implémentation. Mais cela n'est pas satisfaisant et disparaîtra dans de futures versions.

Voir [use](#).

Exemples :

```
# tri alphabétique
@articles = sort @files;
# idem mais avec une routine de tri explicite
@articles = sort {$a cmp $b} @files;
# idem mais indépendant de la casse
@articles = sort {uc($a) cmp uc($b)} @files;
# idem mais dans l'ordre inverse
@articles = sort {$b cmp $a} @files;
# tri numérique ascendant
@articles = sort {$a <=> $b} @files;
```

perlfunc – Fonctions Perl prédéfinies

```
# tri numérique descendant
@articles = sort {$b <=> $a} @files;
# tri de %age par valeur plutôt que par clé
# en utilisant une fonction en-ligne
@eldest = sort { $age{$b} <=> $age{$a} } keys %age;
# tri utilisant le nom explicite d'une subroutine
sub byage {
    $age{$a} <=> $age{$b}; # supposé numérique
}
@sortedclass = sort byage @class;
sub backwards { $b cmp $a; }
@harry = qw(dog cat x Cain Abel);
@george = qw(gone chased yz Punished Axed);
print sort @harry;
    # affiche AbelCaincatdogx
print sort backwards @harry;
    # affiche xdogcatCainAbel
print sort @george, 'to', @harry;
    # affiche AbelAxedCainPunishedcatchaseddoggonetoxyz
# tri inefficace par ordre numérique descendant utilisant
# le premier entier après le signe = ou l'ensemble de
# l'enregistrement si cet entier n'existe pas
@new = sort {
    ($b =~ /=(\d+)/)[0] <=> ($a =~ /=(\d+)/)[0]
        ||
        uc($a) cmp uc($b)
} @old;
# la même chose mais plus efficace;
# nous construisons un tableau auxiliaire d'indices
# pour aller plus vite
@nums = @caps = ();
for (@old) {
    push @nums, /=(\d+)/;
    push @caps, uc($_);
}
@new = @old[ sort {
    $nums[$b] <=> $nums[$a]
        ||
    $caps[$a] cmp $caps[$b]
    } 0..$#old
];
# même chose sans utiliser de variables temporaires
@new = map { $_->[0] }
    sort { $b->[1] <=> $a->[1]
        ||
        $a->[2] cmp $b->[2]
    } map { [$_, /=(\d+)/, uc($_)] } @old;
# l'utilisation d'un prototype vous permet d'utiliser
# n'importe quelle subroutine de comparaison comme
# subroutine de tri (y compris des subroutines d'autres packages)
package other;
sub backwards ($$) { $_[1] cmp $_[0]; } # $a and $b are not set here
package main;
@new = sort other::backwards @old;
# tri stable, sans choix de l'algorithme
use sort 'stable';
@new = sort { substr($a, 3, 5) cmp substr($b, 3, 5) } @old;
# utilisation forcée d'un mergesort (non portable en dehors de Perl 5.8)
use sort '_mergesort'; # notez le _ décourageant
@new = sort { substr($a, 3, 5) cmp substr($b, 3, 5) } @old;
```

Si vous utilisez strict, vous *NE DEVEZ PAS* déclarer \$a et \$b comme variables lexicales. Ce sont des variables globales au package. Cela signifie que si vous êtes dans le package main, c'est :

```
@articles = sort {$main::b <=> $main::a} @files;
```

ou juste :

```
@articles = sort {$::b <=> $::a} @files;
```

mais si vous êtes dans le package FooPack, <c'est :>

```
@articles = sort {$FooPack::b <=> $FooPack::a} @files;
```

La fonction de comparaison doit se comporter correctement. Si elle retourne des résultats incohérents (parfois elle dit que \$x[1] est plus petit que \$x[2] et d'autres fois le contraire par exemple), le résultat du tri n'est pas bien défini.

splice TABLEAU,OFFSET,LONGUEUR,LISTE

splice TABLEAU,OFFSET,LONGUEUR

splice TABLEAU,OFFSET

splice TABLEAU

Supprime d'un tableau les éléments consécutifs désignés par OFFSET (indice du premier élément) et LONGUEUR (nombre d'éléments concernés) et les remplace par les éléments de LISTE si il y en a. Dans un contexte de liste, renvoie les éléments supprimés du tableau. Dans un contexte scalaire, renvoie le dernier élément supprimé ou undef si aucun élément n'est supprimé. Le tableau grossit ou diminue si nécessaire. Si OFFSET est négatif, il est compté à partir de la fin du tableau. Si LONGUEUR est omis, supprime tout à partir de OFFSET. Si LONGUEUR est négatif, supprime les éléments à partir de OFFSET en laissant -LONGUEUR éléments à la fin du tableau. Si OFFSET et LONGUEUR sont omis, supprime tout ce qui est dans le tableau. Si OFFSET va au-delà de la fin du tableau, perl produit un message d'avertissement et splice agit à la fin du tableau.

Les équivalences suivantes sont vraies en supposant que \$[== 0 :

```
push(@a,$x,$y)      splice(@a,@a,0,$x,$y)
pop(@a)             splice(@a,-1)
shift(@a)           splice(@a,0,1)
unshift(@a,$x,$y)  splice(@a,0,0,$x,$y)
$a[$x] = $y        splice(@a,$x,1,$y)
```

Exemple, en supposant que la longueur des tableaux est passée avant chaque tableau :

```
sub aeq { # compare deux listes de valeurs
    my(@a) = splice(@_,0,shift);
    my(@b) = splice(@_,0,shift);
    return 0 unless @a == @b; # même longueur ?
    while (@a) {
        return 0 if pop(@a) ne pop(@b);
    }
    return 1;
}
if (&aeq($len,@foo[1..$len],0+@bar,@bar)) { ... }
```

split /MOTIF,EXPR,LIMITE

split /MOTIF,EXPR

split /MOTIF/

split

Découpe une chaîne en une liste de chaînes et la retourne. Par défaut, les champs vides du début sont conservés et ceux de la fin sont éliminés.

Si l'appel n'est pas dans un contexte de liste, split retourne le nombre de champs trouvés et les place dans le tableau @_. (Dans un contexte de liste, vous pouvez forcer l'utilisation du tableau @_ en utilisant ?? comme motif délimiteur mais il renvoie encore la liste des valeurs.) En revanche, l'utilisation implicite de @_ par split est désapprouvée parce que cela

écrase les arguments de votre subroutine.

SI `EXPR` est omis, `split` découpe la chaîne `$_`. Si `MOTIF` est aussi omis, il découpe selon les blancs (après avoir sauté d'éventuels blancs au départ). Tout ce qui reconnu par `MOTIF` est considéré comme étant un délimiteur de champs. (Remarquez que le délimiteur peut être plus long qu'un seul caractère.)

Si `LIMITE` est spécifié et positif, fixe le nombre maximum de champs du découpage (le nombre de champs dépend du nombre de fois où le `MOTIF` sera reconnu dans `EXPR`). Si `LIMITE` n'est pas spécifié ou vaut zéro, les champs vides de la fin sont supprimés (choses dont les utilisateurs potentiels de `pop()` devraient se souvenir). Si `LIMITE` est négatif, il est traité comme si `LIMITE` avait une valeur arbitrairement très grande. Notez que découper une `EXPR` dont la valeur est la chaîne vide retourne toujours une liste vide, que `LIMIT` soit spécifié ou non.

Un motif qui peut correspondre à la chaîne vide (ne pas confondre avec le motif vide `//` qui n'est qu'un motif parmi tous ceux qui peuvent correspondre à la chaîne vide) découpera la valeur de `EXPR` en caractère séparé à chaque point où il sera reconnu. Par exemple :

```
print join(':', split(/ */, 'hi there'));
```

produira la sortie `h:i:t:h:e:r:e'`.

Un champ vide de début (ou de fin) n'est produit que lorsqu'il y a reconnaissance du `MOTIF` avec une longueur positive au début (à la fin) de la chaîne. Une reconnaissance de longueur nulle en début ou en fin de chaîne ne produira pas de champs vide. Par exemple :

```
print join(':', split(/(=?\w)/, 'hi there!'));
```

produira la sortie `h:i :t:h:e:r:e!'`.

Le paramètre `LIMITE` peut être utilisé pour découper partiellement une ligne :

```
($login, $passwd, $remainder) = split(/:/, $_, 3);
```

Lors de l'affectation à une liste, si `LIMITE` est omis, Perl agit comme si `LIMITE` était égal au nombre de variables de la liste plus un pour éviter tout travail inutile. Pour la liste ci-dessous, la valeur par défaut de `LIMITE` serait 4. Dans les applications où le temps est critique, il vous incombe de ne pas découper en plus de champs que ceux réellement nécessaires.

Si `MOTIF` contient des parenthèses (et donc des sous-motifs), un élément supplémentaire est créé dans le tableau résultat pour chaque chaîne reconnue par le sous-motif.

```
split(/[,-]/, "1-10,20", 3);
```

produit la liste de valeurs

```
(1, '-', 10, ',', 20)
```

Si vous avez dans la variable `$header` tout l'en-tête d'un email normal d'UNIX, vous devriez pouvoir le découper en champs et valeurs en procédant comme suit :

```
$header =~ s/\n\s+/ /g; # fix continuation lines
%hdrs = (UNIX_FROM => split /^(\S*?):\s*/m, $header);
```

Le motif `/MOTIF/` peut être remplacé par une expression pour spécifier un motif qui varie à chaque passage. (Pour faire une compilation une seule fois lors de l'exécution, utilisez `/$variable/o`.)

Un cas spécial : spécifier un blanc (`' '`) comme `MOTIF` découpe selon les espaces exactement comme le fait `split()` sans argument. Donc, `split(' ')` peut être utilisé pour émuler le comportement par défaut de `awk` alors que `split(/ /)` vous donnera autant de champs vides que d'espaces au début. Un `split()` avec `/\s+/` est comme `split(' ')` sauf dans le cas de blancs au début qui produiront un premier champ vide. Un `split()` sans argument effectue réellement un `split(' ', $_)` en interne.

Un `MOTIF /^/` sera traité comme si c'était `/^/m` sinon il ne serait d'aucune utilité.

Exemple :

```
open(PASSWD, '/etc/passwd');
while (<PASSWD>) {
    chomp;
    ($login, $passwd, $uid, $gid,
```

perlfunc – Fonctions Perl prédéfinies

```
$gcos, $home, $shell) = split(/:/);  
#...  
}
```

Comme dans le cas de la reconnaissance d'expressions rationnelles, tout sous-groupe mémorisé non reconnu durant le `split()` produira un `undef` :

```
@fields = split /(A)|B/, "1A2B3";  
# @fields est (1, 'A', 2, undef, 3)
```

sprintf *FORMAT, LISTE*

Retourne une chaîne formatée selon les conventions usuelles de la fonction `sprintf()` de la bibliothèque C. Voir ci-dessous et `sprintf(3)` ou `printf(3)` sur votre système pour une explication sur les principes généraux.

Par exemple :

```
# Produire un nombre avec jusqu'à 8 zéros devant  
$result = sprintf("%08d", $number);  
  
# Arrondir un nombre à la troisième décimale  
$rounded = sprintf("%.3f", $number);
```

Perl a sa propre implémentation de `sprintf()` -- elle émule la fonction C `sprintf()` mais elle ne l'utilise pas (sauf pour les nombres en virgule flottante, et encore en n'autorisant que les modificateurs standards). Conséquence : une extension non standard de votre version locale de `sprintf()` ne sera pas disponible en Perl.

Au contraire de `printf`, `sprintf` ne fera probablement pas ce que vous voulez si vous lui passez un tableau comme premier argument. Ce tableau sera évalué dans un contexte scalaire et, au lieu d'utiliser le premier élément du tableau comme `FORMAT`, Perl utilisera le nombre d'éléments de ce tableau comme `FORMAT`, ce qui n'est pas vraiment utilisable.

Le `sprintf()` de Perl autorise les conversions universellement connues :

```
% un signe pourcent  
%c un caractère dont on fournit le code  
%s une chaîne  
%d un entier signé, en décimal  
%u un entier non-signé, en décimal  
%o un entier non-signé, en octal  
%x un entier non-signé, en hexadécimal  
%e un nombre en virgule flottante, en notation scientifique  
%f un nombre en virgule flottante, avec un nombre de décimales fixe  
%g un nombre en virgule flottante, %e ou %f (au mieux)
```

De plus, Perl autorise les conversions largement supportées :

```
%X comme %x mais avec des lettres majuscules  
%E comme %e, mais en utilisant un "E" majuscule  
%G comme %g, mais en utilisant un "E" majuscule (si nécessaire)  
%b un entier non signé, en binaire  
%p un pointeur (affiche la valeur Perl de l'adresse en hexadécimal)  
%n spécial: stocke le nombre de caractères produits dans la  
prochaine variable de la liste des paramètres
```

Et finalement, pour des raisons de compatibilité (et ``uniquement'' pour cela), Perl autorise les conversions inutiles mais largement supportées :

```
%i un synonyme de %d  
%D un synonyme de %ld  
%U un synonyme de %lu  
%O un synonyme de %lo  
%F un synonyme de %f
```

Notez que le nombre de chiffres utilisés pour l'exposant en notation scientifique produit par `%e`, `%E`, `%g` et `%G` lorsque cet exposant est inférieur à 100 dépend du système : ça peut être 3 chiffres ou moins (avec d'éventuels zéros initiaux). En d'autres termes, 1,23 multiplié par 10 à la puissance 99 peut être ``1.23e99'' ou ``1.23e099''.

Entre le % et la lettre de format, vous pouvez spécifier un certain nombre d'attributs supplémentaires qui permettent de contrôler l'interprétation du format. Dans l'ordre, on trouve :

un index de choix de paramètre

Un index de choix explicite de paramètre tel que 2\$. Par défaut, sprintf formatera le prochain argument inutilisé de la liste mais cela vous permet de choisir votre argument sans respecter cet ordre. Exemple :

```
printf '%2$d %1$d', 12, 34;      # affiche "34 12"
printf '%3$d %d %1$d', 1, 2, 3; # affiche "3 1 1"
```

des flags

un ou plusieurs flags parmi :

```
espace  précède les nombres positifs par un espace
+       précède les nombres positifs par un signe plus
-       justifie le champ à gauche
0       utilise des zéros à la place des espaces pour justifier à droite
#       précède le nombre non nul en octal par "0",
        en hexadécimal par "0x" et en binaire par "0b"
```

Par exemple :

```
printf '<% d>', 12;      # affiche "< 12>"
printf '<+%d>', 12;      # affiche "<+12>"
printf '<%6s>', 12;      # affiche "<      12>"
printf '<%-6s>', 12;     # affiche "<12      >"
printf '<%06s>', 12;     # affiche "<000012>"
printf '<%#x>', 12;      # affiche "<0xc>"
```

des flags vectoriels

Le flag vectoriel et optionnel `v` spécifie le chaîne de jointure à utiliser. Ce flag indique à perl d'interpréter le chaîne fournie comme un vecteur d'entier, un pour chaque caractère dans la chaîne, séparé par un chaîne donnée (un point `.` par défaut). C'est pratique pour afficher des valeurs ou des caractères ordinaux dans des chaînes quelconques :

```
printf "version is v%vd\n", $^V; # La version de Perl
```

Placez un astérisque `*` avant le `v` pour changer la chaîne utilisée comme séparateur de nombres :

```
printf "address is %*vX\n", ":", $addr; # adresse IPv6
printf "bits are %0*v8b\n", " ", $bits; # une chaîne de bits quelconque
```

Vous pouvez aussi spécifier explicitement le numéro d'argument à utiliser comme chaîne de séparateur :

```
printf '%*4$vX%*4$vX%*4$vX', @addr[1..3], ":"; # 3 adresses IPv6
```

largeur (minimale)

Les arguments sont habituellement formatés pour utiliser juste la place nécessaire à la représentation de la valeur. Vous pouvez forcer la largeur en plaçant directement un nombre ou en obtenant la largeur depuis l'argument suivant (via `*`) ou même depuis un argument spécifique (via `*2$`) :

```
printf '<%s>', "a";      # affiche "<a>"
printf '<%6s>', "a";      # affiche "<      a>"
printf '<%*s>', 6, "a";   # affiche "<      a>"
printf '<%*2$s>', "a", 6; # affiche "<      a>"
printf '<%2s>', "long";   # affiche "<long>" (ne tronque pas)
```

Si la largeur obtenue par `*` est négative, cela a le même effet que le flag `-` : une justification à gauche.

précision, ou largeur maximale

perlfunc – Fonctions Perl prédéfinies

Vous pouvez préciser la précision (pour les conversions numériques) ou la largeur maximale (pour les conversions de chaînes) en spécifiant un `.` suivi d'un nombre. Pour les nombres en virgule flottante, cela indique le nombre de décimales à afficher. Exemple :

```
# ces exemples peuvent varier selon le système
printf '<%f>', 1;      # affiche "<1.000000>"
printf '<%.1f>', 1;   # affiche "<1.0>"
printf '<%.0f>', 1;   # affiche "<1>"
printf '<%e>', 10;    # affiche "<1.000000e+01>"
printf '<%.1e>', 10;  # affiche "<1.0e+01>"
```

Pour les nombres entiers, la spécification d'une précision indique la largeur du nombre à afficher. Il sera éventuellement précédé de zéros pour atteindre cette largeur :

```
printf '<%.6x>', 1;      # affiche "<000001>"
printf '<%#.6x>', 1;    # affiche "<0x000001>"
printf '<%-10.6x>', 1;  # affiche "<000001      >"
```

Pour les conversions de chaînes, la spécification d'une précision tronquera la chaîne pour qu'elle tienne dans cette longueur maximale :

```
printf '<%.5s>', "truncated"; # affiche "<trunc>"
printf '<%10.5s>', "truncated"; # affiche "<      trunc>"
```

Vous pouvez aussi récupérer la valeur de précision dans le prochain argument via `.*` :

```
printf '<%.6x>', 1;      # affiche "<000001>"
printf '<%.*x>', 6, 1;    # affiche "<000001>"
```

Ce n'est pas encore faisable mais une future version proposera de récupérer la précision dans un argument choisi en utilisant par exemple `.*2$` :

```
printf '<%.*2$x>', 1, 6;  # INVALIDE, mais affichera un jour "<000001>"
```

taille

Pour les conversions numériques, vous pouvez préciser la taille à utiliser pour interpréter le nombre en utilisant `l`, `h`, `V`, `q`, `L` ou `ll`. Pour les conversions d'entiers, on suppose que la taille est celle utilisée par défaut par votre système pour un entier (habituellement 32 ou 64 bits). Mais vous pouvez changer cela en précisant vous-même un type standard C (l'un de ceux connus du compilateur ayant compilé Perl) à utiliser à la place :

```
l          interprète un entier comme étant
           du type C "long" ou "unsigned long"
h          interprète un entier comme étant
           du type C "short" ou "unsigned short"
q, L ou ll interprète un entier comme étant
           du type C "long long" ou "unsigned long long"
           (si votre plateforme reconnaît ces types, sinon
           c'est une erreur)
```

Pour les conversions de nombres à virgule flottante, on suppose que le nombre est du type utilisé par défaut sur votre système (double ou long double). Mais vous pouvez contraindre l'usage des 'long double' grâce à `q`, `L` ou `ll` si votre plateforme connaît ce type.

Le spécificateur de taille 'V' n'a aucun effet pour le code Perl mais il est accepté pour des raisons de compatibilité avec du code XS ; il signifie « utilise la taille standard pour un entier (ou un nombre en virgule flottante) Perl », ce qui est déjà le cas par défaut dans le code Perl.

l'ordre des arguments

Normalement, `sprintf` utilise le premier argument encore inutilisé comme valeur à

formater pour chaque spécification de format. Si cette spécification utilise * pour utiliser des arguments supplémentaires, ils sont pris dans la liste d'arguments dans l'ordre où ils apparaissent dans la spécification *avant* la valeur elle-même. Lorsqu'un argument est spécifié en utilisant un index explicite, cela ne modifie en rien l'ordre d'utilisation normal des arguments.

Donc :

```
printf '<%. *s>', $a, $b, $c;
```

devrait utiliser \$a pour la largeur, \$b pour la précision et \$c comme valeur à formater alors que :

```
print '<%. 1$. *s>', $a, $b;
```

devrait utiliser \$a comme largeur et comme précision et \$b comme valeur à formater.

Voici quelques exemples supplémentaires – notez bien que pour utiliser les index explicites, il faut parfois protéger le \$:

```
printf "%2\$d %d\n", 12, 34; # affichera "34 12\n"
printf "%2\$d %d %d\n", 12, 34; # affichera "34 12 34\n"
printf "%3\$d %d %d\n", 12, 34, 56; # affichera "56 12 34\n"
printf "%2\$*3\$d %d\n", 12, 34, 3; # affichera " 34 12\n"
```

Si `use locale` est actif, le caractère utilisé comme séparateur décimal pour les nombres réels dépend de la valeur de `LC_NUMERIC`. Voir [the perllocale manpage](#).

sqrt* *EXPR ***sqrt***

Renvoie la racine carrée de *EXPR*. Si *EXPR* est omis, retourne la racine carrée de `$_`. Ne fonctionne que sur les opérandes non négatifs à moins que vous n'ayez chargé le module standard `Math::Complex`.

```
use Math::Complex;
print sqrt(-2); # affiche 1.4142135623731i
```

srand* *EXPR ***srand***

Fixe la graine aléatoire pour l'opérateur [rand\(\)](#).

La seule utilité de cette fonction est d'initialiser la fonction [rand](#) pour que cette dernière produise une séquence différente à chaque exécution de votre programme.

Si [srand\(\)](#) n'est pas appelé explicitement, il l'est implicitement lors de la première utilisation de l'opérateur [rand\(\)](#). Par contre, ce n'était pas le cas dans les versions de Perl antérieures à la version 5.004 et donc, si votre script doit pouvoir tourner avec de vieilles versions de Perl, il doit appeler [srand\(\)](#).

La plupart des programmes n'ont pas besoins d'appeler `srand()`. Seuls le feront ceux ayant besoin d'une graine cryptographiquement forte puisqu'ils ne peuvent se satisfaire de la valeur par défaut calculée en fonction de l'heure courante, du numéro de processus et de l'allocation mémoire ou en fonction du device `/dev/urandom` lorsqu'il est disponible.

Vous pouvez appeler [srand\(\\$graine\)](#) avec la même `$graine` pour que [rand\(\)](#) reproduise la *même* séquence mais cela est réservé à la production de données prévisibles afin de faciliter les tests et le débogage.

N'appellez **pas** [srand\(\)](#) (sans argument) plus d'une fois par exécution de votre script. L'état interne du générateur de nombres aléatoires devrait contenir plus d'entropie que celle fournie par une graine quelconque. Donc un nouvel appel à [srand\(\)](#) ne peut qu'amener à *perdre* de l'aléa.

La plupart des implémentations de [srand](#) utilise un entier et ignoreront silencieusement une éventuelle partie décimale. Cela signifie que l'appel [srand\(42\)](#) produira le même résultat

que l'appel `srand(42.1)`. Pour être correct, passez toujours un entier à `srand`. Dans les versions de Perl antérieures à la version 5.004, la valeur par défaut était juste `time()`. Ce n'est pas une graine particulièrement bonne et donc de nombreux programmes anciens fournissaient leur propre valeur de graine (souvent `time ^ $$` ou `time ^ ($$ + ($$ << 15))`) mais ce n'est plus nécessaire maintenant.

Remarque : vous devez utiliser quelque chose de beaucoup plus aléatoire que la graine par défaut pour des applications de cryptographie. Le checksum d'une sortie compressée d'un ou plusieurs programmes système dont les valeurs changent rapidement est une méthode usuelle. Par exemple :

```
srand (time ^ $$ ^ unpack "%L*", `ps axww | gzip`);
```

Si cela vous intéresse tout particulièrement, regardez le module CPAN

`Math::TrulyRandom`.

Les programmes fréquemment utilisés (comme des scripts CGI) qui utilisent simplement :

```
time ^ $$
```

comme graine peuvent tomber sur la propriété mathématique suivante :

```
a^b == (a+1)^(b+1)
```

une fois sur trois. Donc ne faites pas ça.

stat DESCRIPTEUR

stat EXPR

stat

Retourne une liste de 13 éléments donnant des informations sur l'état soit du fichier dont le nom est donné par EXPR soit du fichier ouvert par DESCRIPTEUR. Si EXPR est omis, retourne l'état de `$_`. Retourne une liste vide en cas d'échec. Typiquement utilisé de la manière suivante :

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

Certains champs ne sont pas gérés par certains types de systèmes de fichiers. Voici la signification de ces champs :

0 dev	numéro de device du système de fichiers
1 ino	numéro d'inode
2 mode	droits du fichier (type et permissions)
3 nlink	nombre de liens (hard) sur le fichier
4 uid	ID numérique de l'utilisateur propriétaire du fichier
5 gid	ID numérique du groupe propriétaire du fichier
6 rdev	l'identificateur de device (fichiers spéciaux uniquement)
7 size	taille totale du fichier, en octets
8 atime	date de dernier accès en secondes depuis l'origine des temps
9 mtime	date de dernière modification en secondes depuis l'origine des temps
10 ctime	date de dernière modification de l'inode en secondes depuis l'origine des temps (*)
11 blksize	taille de blocs préférée pour les E/S sur fichiers
12 blocks	nombre de blocs réellement occupés

(Sur la plupart des systèmes, l'origine des temps est fixée au 1er janvier 1970 à minuit GMT.)

(*) Le champ `ctime` n'est pas portable. En particulier, ne comptez pas dessus pour représenter une « date de création ». Voir [Fichiers in the perlport manpage](#) pour plus de détails.

Si vous passez à `stat` le descripteur spécial dont le nom est le caractère souligné seul, aucun appel à `stat` n'est effectué par contre le contenu courant de la structure d'état du dernier appel à `stat` ou du dernier test de fichier est retourné. Exemple :

```
if (-x $file && (($d) = stat(_)) && $d < 0) {
    print "$file is executable NFS file\n";
}
```

```
}
```

(Ceci ne marche que sur les machines dont le numéro de device est négatif sous NFS.)
Comme le mode contient à la fois le type de fichier et les droits d'accès, vous devrez masquer la portion concernant le type de fichier et utiliser (s)printf avec le format "%o" pour voir les véritables permissions :

```
$mode = (stat($filename))[2];
printf "Permissions are %04o\n", $mode & 07777;
```

Dans un contexte scalaire, `stat()` retourne une valeur booléenne indiquant le succès ou l'échec et positionne, en cas de succès, les informations lié au descripteur spécial `_`.

Le module `File::stat` fournit un mécanisme pratique d'accès par nom :

```
use File::stat;
$sb = stat($filename);
printf "File is %s, size is %s, perm %04o, mtime %s\n",
    $filename, $sb->size, $sb->mode & 07777,
    scalar localtime $sb->mtime;
```

Vous pouvez importer les constantes symboliques de permissions (`S_IF*`) et les fonctions de test (`S_IS*`) depuis le module `Fcntl` :

```
use Fcntl ':mode';
$mode = (stat($filename))[2];
$user_rwx = ($mode & S_IRWXU) >> 6;
$group_read = ($mode & S_IRGRP) >> 3;
$other_execute = $mode & S_IXOTH;
printf "Permissions are %04o\n", S_ISMODE($mode), "\n";
$is_setuid = $mode & S_ISUID;
$is_setgid = S_ISDIR($mode);
```

Vous pourriez écrire ces deux derniers exemples en utilisant les opérateurs `-u` et `-d`. Les constantes communes disponibles sont :

```
# Droits : lecture, écriture, exécution,
# pour utilisateur, groupe, autres.
S_IRWXU S_IRUSR S_IWUSR S_IXUSR
S_IRWXG S_IRGRP S_IWGRP S_IXGRP
S_IRWXO S_IROTH S_IWOTH S_IXOTH
# Setuid/Setgid/Stickness.
S_ISUID S_ISGID S_ISVTX S_ISTXT
# File types. Not necessarily all are available on your system.
S_IFREG S_IFDIR S_IFLNK S_IFBLK S_ISCHR S_IFIFO S_IFSOCK S_IFWHT S_ENFMT
# The following are compatibility aliases for S_IRUSR, S_IWUSR, S_IXUSR.
S_IREAD S_IWRITE S_IEXEC
```

et les fonctions `S_IF*` <sont :>

```
S_IFMODE($mode)    the part of $mode containing the permission bits
                   and the setuid/setgid/sticky bits
S_IFMT($mode)     the part of $mode containing the file type
                   which can be bit-anded with e.g. S_IFREG
                   or with the following functions
# The operators -f, -d, -l, -b, -c, -p, and -s.
S_ISREG($mode) S_ISDIR($mode) S_ISLNK($mode)
S_ISBLK($mode) S_ISCHR($mode) S_ISFIFO($mode) S_ISSOCK($mode)
# No direct -X operator counterpart, but for the first one
# the -g operator is often equivalent. The ENFMT stands for
# record flocking enforcement, a platform-dependent feature.
S_IENFMT($mode) S_ISWHT($mode)
```

Voir la documentation native de `chmod(2)` et de `stat(2)` pour de meilleures informations au sujet des constantes `S_*`.

study SCALAIRE **study**

Prend du temps supplémentaire pour étudier (`study`) la chaîne SCALAIRE (ou `$_` si SCALAIRE est omis) afin d'anticiper de nombreuses recherches d'expressions rationnelles sur elle avant sa prochaine modification. Cela peut améliorer ou non le temps de recherche selon la nature et le nombre de motifs que vous recherchez et selon la fréquence de distribution des caractères dans la chaîne -- vous devriez probablement comparer les temps d'exécution avec et sans pour savoir quand cela est plus rapide. Les boucles qui recherchent de nombreuses petites chaînes constantes (même celles comprises dans des motifs plus complexes) en bénéficient le plus. Il ne peut y avoir qu'un seul `study()` actif à la fois -- si vous étudiez un autre scalaire, le précédent est "oublié". (`study()` fonctionne de la manière suivante : on construit une liste chaînée de tous les caractères de la chaîne à étudier ce qui permet de savoir, par exemple, où se trouve tous les 'k'. Dans chaque chaîne recherchée, on choisit le caractère le plus rare en se basant sur une table de fréquences construite à partir de programmes C et de textes anglais. Seuls sont examinés les endroits qui contiennent ce caractère "rare".)

Par exemple : voici la boucle qui insère une entrée d'index devant chaque ligne contenant un certain motif :

```
while (<>) {
    study;
    print ".IX foo\n" if /\bfoo\b/;
    print ".IX bar\n" if /\bbar\b/;
    print ".IX blurfl\n" if /\bblurfl\b/;
    # ...
    print;
}
```

Lors de la recherche de `/\bfoo\b/`, seuls sont examinés les endroits de `$_` contenant `f` parce que `f` est plus rare que `o`. En général, le gain est important sauf dans des cas pathologiques. Savoir si l'étude initiale est moins coûteuse que le temps gagné lors de la recherche est la seule vraie question.

Remarquez que si vous faites une recherche sur des chaînes que vous ne connaissez que lors de l'exécution, vous pouvez alors construire une boucle entière dans une chaîne que vous évaluez via `eval()` afin d'éviter de recompiler vos motifs à chaque passage.

Combiné avec l'affectation de `undef` à `$/` pour lire chaque fichier comme un seul enregistrement, cela peut être extrêmement rapide et parfois même plus rapide que des programmes spécialisés comme `fgrep(1)`. Le code suivant recherche une liste de mots (`@words`) dans une liste de fichiers (`@files`) et affiche la liste des fichiers qui contiennent ces mots :

```
$search = 'while (<>) { study;';
foreach $word (@words) {
    $search .= "++\${seen}{\${ARGV}} if /\b${word}\b/;\n";
}
$search .= "}";
@ARGV = @files;
undef $/;
eval $search;           # ca parle...
$/ = "\n";             # retour au délimiteur de ligne normal
foreach $file (sort keys(%seen)) {
    print $file, "\n";
}
```

sub NOM BLOC

sub NOM (PROTO) BLOC

sub NOM : ATTRS BLOC

sub NOM (PROTO) : ATTRS BLOC

C'est la définition de sous-routine. Pas vraiment une fonction en tant que telle. Avec juste le nom NOM (et un éventuel prototype), c'est une simple déclaration préalable. Sans NOM, c'est la déclaration d'une fonction anonyme et cela retourne une valeur : la référence du CODE de la fermeture que vous venez de créer.

Voir [the perlsub manpage](#) et [the perlref manpage](#) pour plus de détails sur les sous-routines et les références et [attributes](#) et [the Attribute::Handlers manpage](#) pour plus de détails au sujet des attributs (ATTRS).

substr* *EXPR,OFFSET,LONGUEUR,REPLACEMENT

substr* *EXPR,OFFSET,LONGUEUR

substr* *EXPR,OFFSET

Extrait une sous-chaîne de EXPR et la retourne. Le premier caractère est à l'indice 0 ou à ce que vous avez fixé par \$[(mais ne le faites pas). Si OFFSET est négatif (ou plus précisément plus petit que \$[), le compte a lieu à partir de la fin de la chaîne. Si LONGUEUR est omis, retourne tous les caractères jusqu'à la fin de la chaîne. Si LONGUEUR est négatif, il indique le nombre de caractères à laisser à la fin de la chaîne. Vous pouvez utiliser la fonction [substr\(\)](#) comme une lvalue auquel cas EXPR doit aussi être une lvalue. Si vous affectez quelque chose de plus court que LONGUEUR, la chaîne raccourcit et si vous affectez quelque chose de plus long que LONGUEUR, la chaîne grossit. Pour conserver la même longueur vous pouvez remplir ou couper votre valeur en utilisant [sprintf\(\)](#).

Si OFFSET et LONGUEUR spécifient une sous-chaîne qui est partiellement en dehors de la chaîne, seule la partie qui est dans la chaîne qui est retournée. Si la sous-chaîne est entièrement en dehors de la chaîne, un message d'avertissement (warning) est produit et la valeur undef est retournée. Lorsque [substr\(\)](#) est utilisé en tant que lvalue, spécifier une sous-chaîne entièrement en dehors de la chaîne produit une erreur fatale. Voici un exemple illustrant ce comportement :

```
my $name = 'fred';
substr($name, 4) = 'dy';           # $name vaut maintenant 'freddy'
my $null = substr $name, 6, 2;    # retourne '' (sans avertissement)
my $oops = substr $name, 7;      # retourne undef, avec avertissement
substr($name, 7) = 'gap';        # erreur fatale
```

Un autre moyen d'utiliser [substr\(\)](#) comme lvalue est de spécifier la chaîne de remplacement comme quatrième argument (REPLACEMENT). Ceci permet de remplacer une partie de la chaîne en récupérant ce qui y était auparavant en une seule opération exactement comme avec [splice\(\)](#).

symlink* *OLDFILE,NEWFILE

Crée un nouveau fichier (NEWFILE) lié symboliquement au vieux fichier (OLDFILE).

Retourne 1 en cas de succès ou 0 autrement. Produit une erreur fatale lors de l'exécution sur les systèmes qui ne supportent pas les liens symboliques. Pour vérifier cela, utilisez eval :

```
$symlink_exists = eval { symlink("", ""); 1 };
```

syscall* *LISTE

Réalise l'appel système spécifié comme premier élément de la liste en lui passant les éléments restants comme arguments. Cela produit une erreur fatale s'il n'est pas implémenté. Les arguments sont interprétés de la manière suivante : si un argument donné est numérique, il est passé comme un entier. Sinon, c'est le pointeur vers la valeur de la chaîne qui est passé. Vous avez la charge de vous assurer qu'une chaîne est déjà assez longue pour recevoir un résultat qui pourrait y être écrit. Vous ne pouvez pas utiliser de chaîne littérale (ou autres chaîne en lecture seule) comme argument de [syscall\(\)](#) parce

que Perl s'assure qu'il est possible d'écrire dans toutes les chaînes dont il passe les pointeurs. Si votre argument numérique n'est pas un littéral et n'a jamais été interprété dans un contexte numérique, vous pouvez lui ajouter 0 pour forcer Perl à le voir comme un nombre. Le code suivant émule la fonction `syswrite()` :

```
require 'syscall.ph'; # peut nécessiter de faire tourner h2ph
$s = "hi there\n";
syscall(&SYS_write, fileno(STDOUT), $s, length $s);
```

Remarquez que Perl ne peut pas passer plus de 14 arguments à votre appel système, ce qui en pratique suffit largement.

`syscall` retourne la valeur retournée par l'appel système appelé. Si l'appel système échoue, `syscall()` retourne -1 et positionne \$! (errno). Remarquez que certains appels systèmes peuvent légitimement retourner -1. Le seul moyen de gérer cela proprement est de faire \$!=0 avant l'appel système et de regarder la valeur de \$! lorsque `syscall` retourne -1.

Il y a un problème avec `syscall(&SYS_pipe)` : cela retourne le numéro du fichier créé côté lecture du tube. Il n'y a aucun moyen de récupérer le numéro de fichier de l'autre côté. Vous pouvez contourner ce problème en utilisant `pipe()` à la place.

***sysopen* DESCRIPTEUR,FILENAME,MODE**

***sysopen* DESCRIPTEUR,FILENAME,MODE,PERMS**

Ouvre le fichier dont le nom est donné par FILENAME et l'associe avec DESCRIPTEUR. Si DESCRIPTEUR est une expression, sa valeur représente le nom du véritable descripteur à utiliser. Cette fonction appelle la fonction `open()` du système sous-jacent avec les paramètres FILENAME, MODE et PERMS.

Les valeurs possibles des bits du paramètre MODE sont dépendantes du système ; elles sont disponibles via le module standard `Fcntl`. Lisez la documentation du `open` de votre système d'exploitation pour savoir quels sont les bits disponibles. Vous pouvez combiner plusieurs valeurs en utilisant l'opérateur |.

Quelques-unes des valeurs les plus courantes sont `O_RDONLY` pour l'ouverture en lecture seule, `O_WRONLY` pour l'ouverture en écriture seule, et `O_RDWR` pour l'ouverture en mode lecture/écriture.

Pour des raisons historiques, quelques valeurs fonctionnent sur la plupart des systèmes supportés par perl : zéro signifie lecture seule, un signifie écriture seule et deux signifie lecture/écriture. Nous savons que ces valeurs *ne* fonctionnent *pas* sous Unix OS/390 ou sur Macintosh ; vous ne devriez sans doute pas les utiliser dans du code nouveau.

Si le fichier nommé FILENAME n'existe pas et que l'appel à `open()` le crée (typiquement parce que MODE inclut le flag `O_CREAT`) alors la valeur de PERM spécifie les droits de ce nouveau fichier. Si vous avez omis l'argument PERM de `sysopen()`, Perl utilise la valeur octale 0666. Les valeurs de droits doivent être fournies en octal et sont modifiées par la valeur courante du `umask` de votre process.

Sur la plupart des systèmes, le flag `O_EXCL` permet d'ouvrir un fichier en mode exclusif. Ce n'est **pas** du verrouillage : l'exclusivité signifie ici que si le fichier existe déjà, `sysopen()` échoue. `O_EXCL` l'emporte sur `O_TRUNC`.

Parfois vous voudrez tronquer un fichier existant : `O_TRUNC`.

Vous devriez éviter d'imposer un mode 0644 comme argument de `sysopen` parce que cela enlève à l'utilisateur la possibilité de fixer un `umask` plus permissif. Voir `umask` pour plus de détails.

Notez que `sysopen` dépend de la fonction `fdopen()` de votre bibliothèque C. Sur de nombreux systèmes UNIX, `fdopen()` est connue pour échouer si le nombre de descripteurs de fichiers excède une certaine valeur, typiquement 255. Si vous avez besoin de plus de descripteurs, pensez à recompiler Perl en utilisant la bibliothèque `sfi` ou à utiliser la fonction `POSIX::open()`.

Voir *perlopentut* pour une initiation à l'ouverture de fichiers.

***sysread* DESCRIPTEUR,SCALAIRE,LONGUEUR,OFFSET**

***sysread* DESCRIPTEUR,SCALAIRE,LONGUEUR**

Tente de lire LONGUEUR caractères de données à partir du DESCRIPTEUR spécifié en utilisant l'appel système `read(2)`. Les données lues sont stockées dans la variable SCALAIRE. Comme cette fonction n'utilise pas les entrées/sorties bufferisées, le mélange avec d'autres sortes de lecture/écriture comme `print`, `write`, `seek`, `tell` ou `eof` peut mal se passer parce que habituellement PerlIO bufferise les données. Retourne le nombre de caractères réellement lus, 0 à la fin du fichier ou undef en cas d'erreur. SCALAIRE grossira ou réduira afin que le dernier octet lu soit effectivement le dernier octet du scalaire après la lecture.

Notez que l'on parle bien de *caractères* : selon l'état du DESCRIPTEUR, ce seront soit des octets (8-bit) soit des caractères qui seront lus. Par défaut tous les DESCRIPTEURS opèrent sur des octets mais si un DESCRIPTEUR a été ouvert en spécifiant le filtre d'entrée/sortie `:utf8` (voir la directive `open` et `open`) alors les entrées/sorties se feront sur des caractères et non sur des octets.

Un OFFSET peut être spécifié pour placer les données lues ailleurs qu'au début de la chaîne. Un OFFSET négatif spécifie un emplacement en comptant les caractères à partir de la fin de la chaîne. Un OFFSET positif plus grand que la longueur de SCALAIRE agrandira la chaîne jusqu'à la taille requise en la remplissant avec des octets `"\0"` avant de lui ajouter le résultat de la lecture.

Il n'y a pas de fonction `syseof()`, ce qui n'est pas un mal puisque `eof()` ne marche pas très bien sur les fichiers device (comme les `tty`). Utilisez `sysread()` et testez une valeur de retour à zéro pour savoir si c'est terminé.

***sysseek* DESCRIPTEUR,POSITION,WHENCE**

Spécifie, en octets, la position système d'un DESCRIPTEUR en utilisant l'appel système `lseek(2)`. DESCRIPTEUR peut être une expression qui donne le nom du descripteur à utiliser. Les valeurs de WHENCE sont 0 pour mettre la nouvelle position à POSITION, 1 pour la mettre à la position courante plus POSITION et 2 pour la mettre à EOF plus POSITION (typiquement une valeur négative).

Notez bien « en octets » : même si le DESCRIPTEUR a été configuré pour opérer sur des caractères (en utilisant le filtre d'entrée/sortie `:utf8` par exemple), `tell()` retournera une position en octets et non en caractères (l'implémentation d'une telle fonctionnalité aurait ralenti énormément `sysseek()`).

`sysseek()` passe outre les tampons habituels d'entrée/sortie. Donc le mélange avec d'autres sortes de lecture/écriture (autre que `sysread()`) comme `print()`, `write()`, `seek()` ou `tell()` peut tout casser.

Pour WHENCE, vous pouvez utiliser les constantes `SEEK_SET`, `SEEK_CUR` et `SEEK_END` (début de fichier, position courante, fin de fichier) provenant du module `Fcntl`. L'usage de ces constantes rend votre script plus portable. Voici un exemple qui définit la fonction `sysstell` :

```
use Fcntl 'SEEK_CUR';
sub sysstell { sysseek($_[0], 0, SEEK_CUR) }
```

Retourne la nouvelle position ou undef en cas d'échec. Une position nulle est retournée par la valeur `"0 but true"` ; Donc `sysseek()` retourne `true` (vrai) en cas de succès et `false` (faux) en cas d'échec et vous pouvez encore déterminer facilement la nouvelle position.

***system* LISTE**

***system* PROGRAMME LISTE**

Fait exactement la même chose que `exec LISTE` sauf qu'un fork est effectué au préalable et que le process parent attend que son fils ait terminé. Remarquez que le traitement des arguments dépend de leur nombre. Si il y a plus d'un argument dans LISTE ou si LISTE est un tableau avec plus d'une valeur, system exécute le programme donné comme premier argument avec comme arguments ceux donnés dans le reste de la liste. Si il n'y a qu'un seul argument dans LISTE et s'il contient des méta-caractères du shell, il est passé en entier au shell de commandes du système pour être interprété (c'est `/bin/sh -c` sur les plates-formes Unix mais cela peut varier sur les autres). Si il ne contient pas de méta-caractères du shell, il est alors découpé en mots et passé directement à `execvp()`, ce qui est plus efficace.

Depuis la version v5.6.0, Perl tente de vider les tampons de tous les fichiers ouverts en écriture avant d'effectuer une opération impliquant un `fork()` mais cela n'est pas supporté sur toutes les plates-formes (voir [the perlport manpage](#)). Pour être plus sûr, vous devriez positionner la variable `$|` (`$AUTOFLUSH` en anglais) ou appelé la méthode `autoflush()` des objets `IO::Handle` pour chacun des descripteurs ouverts.

La valeur retournée est le statut de sortie (exit status) du programme tel que retourné par l'appel `wait()`. Pour obtenir la valeur réelle de sortie, il faut décaler la valeur de retour de 8 bits vers la droite. Voir aussi [exec](#). Ce n'est pas ce qu'il faut utiliser pour capturer la sortie d'une commande. Pour cela, regarder les apostrophes inversées (backticks) ou `qx//` comme décrit dans [`CHAINE` in the perlport manpage](#). Une valeur de retour `-1` indique l'échec du lancement du programme (`#!` en donne la raison).

Comme `exec()`, `system()` vous autorise à définir le nom sous lequel le programme apparaît si vous utilisez la syntaxe `system PROGRAMME LISTE`. Voir [exec](#).

Puisque `system()` et les apostrophes inversées (backticks) bloquent `SIGINT` et `SIGQUIT`, tuer le programme qu'ils font tourner n'interrompra pas votre programme.

```
@args = ("command", "arg1", "arg2");
system(@args) == 0
    or die "system @args failed: $?"
```

Vous pouvez tester tous les cas possibles d'échec en analysant `$?` de la manière suivante :

```
$exit_value = $? >> 8;
$signal_num = $? & 127;
$dumped_core = $? & 128;
```

ou, de manière plus portable, en utilisant les appels `W*()` du module `POSIX`. Voir [the perlport manpage](#) pour plus d'information.

Lorsque les arguments sont exécutés via le shell système, les résultats et codes de retour sont sujet à tous ses caprices et capacités. Voir [`CHAINE` in the perlport manpage](#) et [exec](#) pour plus de détails.

***syswrite* DESCRIPTEUR,SCALAIRE,LONGUEUR,OFFSET**

***syswrite* DESCRIPTEUR,SCALAIRE,LONGUEUR**

***syswrite* DESCRIPTEUR,SCALAIRE**

Essaye d'écrire LONGUEUR caractères provenant de la variable SCALAIRE sur le DESCRIPTEUR spécifié en utilisant l'appel système `write(2)`. Si LONGUEUR n'est pas spécifié, c'est le contenu complet de SCALAIRE qui est écrit. Passe outre la bibliothèque `stdio`. Donc le mélange avec d'autres sortes de lecture/écriture (autre que `sysread()`) comme `print()`, `write()`, `seek()` ou `tell()` peut mal se passer parce que habituellement `stdio` bufferise les données. Retourne le nombre de caractères réellement écrits ou `undef` en cas d'erreur. Si LONGUEUR est plus grand que la quantité de données disponibles dans SCALAIRE après OFFSET, seules les données disponibles sont écrites. Un OFFSET peut être spécifié pour lire les données à écrire à partir d'autre chose que le début du scalaire. Un OFFSET négatif calcule l'emplacement en comptant les caractères à partir de la fin de la chaîne. Au cas où SCALAIRE est vide, vous pouvez utiliser OFFSET

mais uniquement avec la valeur zéro.

Notez que l'on parle bien de *caractères* : selon l'état du DESCRIPTEUR, ce seront soit des octets (8-bit) soit des caractères qui seront lus. Par défaut tous les DESCRIPTEURS opèrent sur des octets mais si un DESCRIPTEUR a été modifié via `binmode()` pour utiliser le filtre d'entrée/sortie `:utf8` (voir la directive `open` dans `open`) alors les entrées/sorties se feront sur des caractères et non sur des octets.

***tell* DESCRIPTEUR**

tell

Retourne la position courante en *octets* de DESCRIPTEUR ou `-1` en cas d'erreur.

DESCRIPTEUR peut être une expression dont la valeur donne le nom du descripteurs réel. Si DESCRIPTEUR est omis, on utilise le dernier fichier lu.

Notez bien « en *octets* » : même si le DESCRIPTEUR a été configuré pour opérer sur des caractères (en utilisant le filtre d'entrée/sortie `:utf8` par exemple), `tell()` retournera une position en octets et non en caractères (l'implémentation d'une telle fonctionnalité aurait ralenti énormément `tell()`).

Il n'y a pas de fonction `sysstell`. Utilisez `sysseek(FH, 0, 1)` à la place.

N'utilisez pas `tell()` sur des DESCRIPTEURS qui ont été ouverts via `sysopen()`. Utilisez `sysseek()` pour cela. Pourquoi ? Parce que `sysopen()` crée des DESCRIPTEURS sans bufferisation alors que `open()` utilise la bufferisation. `sysseek()` n'a un sens que dans le premier cas et `tell()` que dans le second.

***tell* DIRHANDLE**

Retourne la position courante du dernier `readdir()` effectué sur DIRHANDLE. La valeur retournée peut être fournie à `seekdir()` pour accéder à un endroit particulier dans ce répertoire. Pose les mêmes problèmes que l'appel système correspondant.

***tie* VARIABLE, CLASSNAME, LISTE**

Cette fonction relie une variable à une classe d'un package qui fournit l'implémentation de cette variable. VARIABLE est le nom de la variable à relier. CLASSNAME est le nom de la classe implémentant les objets du type correct. Tout argument supplémentaire est passé tel quel à la méthode `new()` de la classe (à savoir `TIESCALAR`, `TIEARRAY` ou `TIEHASH`). Typiquement, ce sont des arguments tels que ceux passés à la fonction `C dbm_open()`. L'objet retourné par la méthode `new()` est aussi retourné par la fonction `tie()` ce qui est pratique si vous voulez accéder à d'autres méthodes de CLASSNAME.

Remarquez que des fonctions telles que `keys()` et `values()` peuvent retourner des listes énormes lorsqu'elles sont utilisées sur de gros objets comme des fichiers DBM. Vous devriez plutôt utiliser la fonction `each()` pour les parcourir. Exemple :

```
# print out history file offsets
use NDBM_File;
tie(%HIST, 'NDBM_File', '/usr/lib/news/history', 1, 0);
while (($key,$val) = each %HIST) {
    print $key, ' = ', unpack('L',$val), "\n";
}
untie(%HIST);
```

Une classe implémentant une table de hachage devrait définir les méthodes suivantes :

```
TIEHASH classname, LISTE
FETCH this, key
STORE this, key, value
DELETE this, key
CLEAR this
EXISTS this, key
FIRSTKEY this
```

perlfunc – Fonctions Perl prédéfinies

```
NEXTKEY this, lastkey
DESTROY this
UNTIE this
```

Une classe implémentant un tableau devrait définir les méthodes suivantes :

```
TIEARRAY classname, LISTE
FETCH this, key
STORE this, key, value
FETCHSIZE this
STORESIZE this, count
CLEAR this
PUSH this, LISTE
POP this
SHIFT this
UNSHIFT this, LISTE
SPlice this, offset, length, LISTE
EXTEND this, count
DESTROY this
UNTIE this
```

Une classe implémentant un descripteur de fichier devrait définir les méthodes suivantes :

```
TIEHANDLE classname, LISTE
READ this, scalar, length, offset
READLINE this
GETC this
WRITE this, scalar, length, offset
PRINT this, LISTE
PRINTF this, format, LISTE
BINMODE this
EOF this
FILENO this
SEEK this, position, whence
TELL this
OPEN this, mode, LIST
CLOSE this
DESTROY this
UNTIE this
```

Une classe implémentant un scalaire devrait définir les méthodes suivantes :

```
TIESCALAR classname, LISTE
FETCH this
STORE this, value
DESTROY this
UNTIE this
```

Il n'est pas absolument nécessaire d'implémenter toutes les méthodes décrites ci-dessus. Voir [the perltie manpage](#), [the Tie::Hash manpage](#), [the Tie::Array manpage](#), [the Tie::Scalar manpage](#), et [the Tie::Handle manpage](#).

Au contraire de [dbmopen\(\)](#), la fonction [tie\(\)](#) n'effectue pas pour vous le 'use' ou le 'require' du module -- vous devez le faire vous-même explicitement. Voir les modules [DB_File](#) ou [Config](#) pour des utilisations intéressantes de [tie\(\)](#).

Pour de plus amples informations, voir [the perltie manpage](#) et [tied VARIABLE](#).

tied VARIABLE

Retourne une référence vers l'objet caché derrière VARIABLE (la même valeur que celle retournée par l'appel [tie\(\)](#) qui a lié cette variable à un package). Retourne la valeur undef si VARIABLE n'est pas lié à un package.

time

Retourne le nombre de secondes écoulées depuis ce que le système considère comme étant l'origine des temps (C'est le 1er janvier 1904 à minuit pour Mac OS et le 1er janvier

1970 à minuit pour la plupart des autres systèmes). Peut servir comme argument de `gmtime()` et de `localtime()`.

Pour mesurer le temps avec une meilleure granularité que la seconde, vous pouvez utiliser le module `Time::HiRes` disponible sur CPAN ou, si vous avez `gettimeofday(2)`, vous pouvez utiliser l'interface `syscall` de Perl. Voir [the perfaq8 manpage](#) pour les détails.

times

Retourne une liste de quatre éléments donnant le temps utilisateur et système en secondes, pour ce process et pour les enfants de ce process.

```
( $user, $system, $cuser, $csystem ) = times;
```

Dans un contexte scalaire `times` retourne `$user`.

tr///

L'opérateur de translittération. La même chose que `y///`. Voir [the perlop manpage](#).

truncate DESCRIPTEUR, LONGUEUR

truncate EXPR, LONGUEUR

Tronque le fichier ouvert par `DESCRIPTEUR` ou nommé par `EXPR` à la taille `LONGUEUR` spécifiée. Produit une erreur fatale si la troncature n'est pas implémentée sur votre système. Retourne `true` (vrai) en cas de succès ou la valeur `undef` autrement.

Le comportement de `truncate()` est indéfini si `LONGUEUR` est plus grand que la longueur actuelle du fichier.

uc EXPR

uc

Retourne la version en majuscule de `EXPR`. C'est la fonction interne qui implémente la séquence d'échappement `\U` des chaînes entre guillemets. Respecte le locale `LC_CTYPE` courant si `use locale` est actif. Voir [the perllocale manpage](#) et [perlunicode](#) pour plus de détails. Ne gère pas la table spécifique aux majuscules apparaissant en début de mot. Voir [ucfirst](#) pour cela.

Si `EXPR` est omis, s'applique à `$_`.

ucfirst EXPR

ucfirst

Retourne la valeur de `EXPR` avec le premier caractère en majuscule. C'est la fonction interne qui implémente la séquence d'échappement `\u` des chaînes entre guillemets.

Respecte le locale `LC_CTYPE` courant si `use locale` est actif. Voir [the perllocale manpage](#) et [perlunicode](#) pour plus de détails.

Si `EXPR` est omis, s'applique à `$_`.

umask EXPR

umask

Positionne le umask du process à `EXPR` et retourne la valeur précédente. Si `EXPR` est omis, retourne la valeur courante du umask.

Les droits Unix `rwxxr-x---` sont représentés par trois ensembles de trois bits ou par trois nombres octaux : `0750` (le 0 initial indique une valeur octale et ne fait pas partie des chiffres). La valeur de `umask` est un tel nombre qui représente les bits de permissions désactivés. Les valeurs de permissions (ou "droits" ou "mode") que vous passez à [mkdir](#) ou [sysopen](#) sont modifiées par votre umask. Par exemple, si vous dites à [sysopen](#) de créer un fichier avec les droits `0777` et si votre umask vaut `0022` alors le fichier sera réellement créé avec les droits `0755`. Si votre umask vaut `0027` (le groupe ne peut écrire ;

les autres ne peuvent ni lire, ni écrire, ni exécuter) alors passer 0666 à `sysopen` créera un fichier avec les droits 0640 (0666 &~ 027 vaut 0640).

Voici quelques conseils : fournissez un mode de création de 0666 pour les fichiers normaux (dans `sysopen`) et de 0777 pour les répertoires (dans `mkdir`) et les fichiers exécutables. Cela donne la liberté à l'utilisateur de choisir : si il veut des fichiers protégés, il peut choisir un umask de 022 ou 027 ou même le umask particulièrement antisocial 077. Dans ce domaine, les programmes peuvent rarement (si ce n'est jamais) prendre de meilleures décisions que l'utilisateur. L'exception concerne les fichiers qui doivent être gardés privé : fichiers de mail, fichiers de cookies des navigateurs, fichiers `.rhosts` et autres.

Si `umask(2)` n'est pas implémenté sur votre système et que vous êtes en train de restreindre les droits d'accès pour *vous-même* (i.e. $(\text{EXPR} \& 0700) > 0$), cela produit une erreur fatale lors de l'exécution. Si `umask(2)` n'est pas implémenté et que vous ne modifiez pas vos propres droits d'accès, retourne `undef`.

Souvenez-vous que umask est un nombre, habituellement donné en octal ; Ce n'est pas une chaîne de chiffres en octal. Voir aussi `oct`, si vous disposez d'une chaîne.

undef EXPR

undef

Donne à EXPR la valeur indéfinie (undef). EXPR doit être une lvalue. À n'utiliser que sur des scalaires, des tableaux (en utilisant @), des tables de hachage (en utilisant %), des sous-routines (en utilisant &) ou des typeglob (en utilisant *). (Dire `undef $hash{$key}` ne fera probablement pas ce que vous espérez sur la plupart des variables prédéfinies ou sur les liste de valeurs DBM. Ne le faites donc pas ; voir `delete`.) Retourne toujours la valeur undef. Vous pouvez omettre EXPR, auquel cas rien ne sera indéfini mais vous récupérerez encore la valeur undef afin, par exemple, de la retourner depuis une sous-routine, de l'affecter à une variable ou de la passer comme argument. Exemples :

```
undef $foo;
undef $bar{'blurfl'};      # À comparer à : delete $bar{'blurfl'};
undef @ary;
undef %hash;
undef &mysub;
undef *xyz;               # détruit $xyz, @xyz, %xyz, &xyz, etc.
return (wantarray ? (undef, $errmsg) : undef) if $they_blew_it;
select undef, undef, undef, 0.25;
($a, $b, undef, $c) = &foo;      # Ignorer la troisième valeur retournée
```

Remarquez que c'est un opérateur unaire et non un opérateur de liste.

unlink LISTE

unlink

Efface une liste de fichiers. Retourne le nombre de fichiers effacés avec succès.

```
$cnt = unlink 'a', 'b', 'c';
unlink @goners;
unlink <*.bak>;
```

Remarque : `unlink()` n'effacera pas de répertoires à moins que vous ne soyez le super-utilisateur (root) et que l'option `-U` soit donnée à Perl. Même si ces conditions sont remplies, soyez conscient que l'effacement d'un répertoire par unlink peut endommager votre système de fichiers. Utilisez `rmdir()` à la place.

Si LISTE est omis, s'applique à \$_.

unpack TEMPLATE,EXPR

`unpack()` réalise l'opération inverse de `pack()` : il prend une chaîne représentant une structure, la décompose en une liste de valeurs et retourne la liste de ces valeurs. (Dans un contexte scalaire, il retourne simplement la première valeur produite.)

La chaîne est découpée en morceaux selon le format TEMPLATE fourni. Chaque morceau est converti séparément en une valeur. Typiquement, soit la chaîne est le résultat d'un pack soit les octets de la chaîne représentent une structure C d'un certain type.

Le TEMPLATE a le même format que pour la fonction pack(). Voici une subroutine qui extrait une sous-chaîne :

```
sub substr {
    my($what, $where, $howmuch) = @_;
    unpack("x$where a$howmuch", $what);
}
```

et un autre exemple :

```
sub ordinal { unpack("c", $_[0]); } # identique à ord()
```

En plus, vous pouvez préfixer un champ avec un %<nombre> pour indiquer que vous voulez un checksum des items sur <nombre> bits à la place des items eux-mêmes. Par défaut, c'est un checksum sur 16 bits. Le checksum est calculé en additionnant les valeurs numériques des valeurs extraites (pour les champs alphanumériques, c'est la somme des ord(\$caractere) qui est prise et pour les champs de bits, c'est la somme des 1 et des 0). Par exemple, le code suivant calcule le même nombre que le programme sum System V :

```
$checksum = do {
    local $/; # slurp!
    unpack("%32C*", <>) % 65535;
};
```

Le code suivant calcule de manière efficace le nombre de bits à un dans un vecteur de bits :

```
$setbits = unpack("%32b*", $selectmask);
```

Les formats p et P doivent être utilisés avec précautions. Puisque Perl n'a aucun moyen de vérifier que les valeurs passées à unpack() correspondent à des emplacements mémoires valides, le passage d'un pointeur dont on n'est pas sûr de la validité peut avoir des conséquences désastreuses.

Si la valeur de répétition d'un champ est plus grande que ce que le reste de la chaîne d'entrée autorise, la valeur est diminuée. Si la chaîne d'entrée est plus longue que ce qui est décrit par TEMPLATE, le reste est ignoré.

Voir pack pour plus d'exemples et de remarques.

untie VARIABLE

Casse le lien entre une variable et un package. (Voir tie().) N'a aucun effet si la variable n'est liée à aucun package.

unshift TABLEAU,LISTE

Fait le contraire de shift(). Ou le contraire de push(), selon le point de vue. Ajoute la liste LISTE au début du tableau TABLEAU et retourne le nouveau nombre d'éléments du tableau.

```
unshift(@ARGV, '-e') unless $ARGV[0] =~ /^-/;
```

Remarquez que LISTE est ajoutée d'un seul coup et non élément par élément. Donc les éléments restent dans le même ordre. Utilisez reverse() pour faire le contraire.

use Module VERSION LISTE

use Module VERSION

use Module LISTE

use Module

use VERSION

Importe dans le package courant quelques ``sémantiques" du module spécifié, généralement en les attachant à certains noms de subroutines ou de variables de votre package. C'est exactement équivalent à :

```
BEGIN { require Module; import Module LISTE; }
```

sauf que Module *doit* être un mot (bareword).

VERSION peut être soit une valeur numérique telle que 5.006 qui sera alors comparée à \$], soit une valeur littérale de la forme v5.6.1 qui sera alors comparée à \$^V (c.-à-d. \$PERL_VERSION). Si VERSION est plus grand que la version courante de l'interpréteur Perl alors il se produira une erreur fatale ; Perl n'essaiera même pas de lire le reste du fichier. À comparer avec `require` qui fait la même vérification mais lors de l'exécution. L'utilisation d'une VERSION sous forme littérale (ex: v5.6.1) devrait être évitée puisque cela entraîne la production de messages d'erreurs erronés par les vieilles versions de Perl qui ne reconnaissent pas cette syntaxe. L'équivalent numérique est préférable.

```
use v5.6.1;      # vérification de version à la compilation
use 5.6.1;      # idem
use 5.005_03;   # numéro de version numérique
                # (préférable pour des raisons de compatibilité)
```

C'est pratique si vous devez vérifier la version courante de Perl avant d'utiliser (via `use`) des modules qui ont changé de manière incompatible depuis les anciennes versions. (Nous essayons de le faire le moins souvent possible).

Le BEGIN force le `require` et le `import` lors de la compilation. Le `require` assure que le module est chargé en mémoire si il ne l'a pas déjà été. `import` n'est pas une fonction interne -- c'est juste un appel à une méthode statique ordinaire dans le package `Module` pour lui demander d'importer dans le package courant la liste de ses fonctionnalités. Le module peut implémenter sa méthode `import()` comme il le veut, bien que la plupart des modules préfèrent simplement la définir par héritage de la classe `Exporter` qui est définie dans le module `Exporter`. Voir `Exporter`. Si aucune méthode `import()` ne peut être trouvée alors l'appel est ignoré.

Si vous voulez éviter l'appel à la méthode `import` d'un package (par exemple pour éviter que votre espace de noms soit modifié), fournissez explicitement une liste vide :

```
use Module ();
```

C'est exactement équivalent à :

```
BEGIN { require Module }
```

Si l'argument VERSION est présent entre Module et LISTE alors `use` appelle la méthode VERSION de la classe Module avec la version spécifiée comme argument. La méthode VERSION par défaut, héritée de la classe `Universal`, crie (via `croak`) si la version demandée est plus grande que celle fournie par la variable `$Module::VERSION`.

À nouveau, il y a une différence entre omettre LISTE (`import` appelé sans argument) et fournir une LISTE explicitement vide `()` (`import` n'est pas appelé). Remarquez qu'il n'y a pas de virgule après VERSION !

Puisque c'est une interface largement ouverte, les pragmas (les directives du compilateur) sont aussi implémentés en utilisant ce moyen. Les pragmas actuellement implémentés sont :

```
use constant;
use diagnostics;
use integer;
use sigtrap qw(SEGV BUS);
use strict qw(subs vars refs);
use subs qw(afunc blurfl);
use warnings qw(all);
use sort qw(stable _quicksort _mergesort);
```

Certains d'entre eux sont des pseudo-modules qui importent de nouvelles sémantiques uniquement dans la portée du bloc courant (comme `strict` or `integer`) contrairement aux modules ordinaires qui importent des symboles dans le package courant (qui sont donc effectifs jusqu'à la fin du fichier lui-même).

Il y a une commande `no` permettant de "désimporter" des choses importées par `use`, i.e. elle appelle `unimport Module LISTE` à la place de `import`.

```
no integer;
no strict 'refs';
no warnings;
```

Voir [the perlmodlib manpage](#) pour une liste des modules et pragmas standard. Voir [the perlrun manpage](#) pour les options `-M` et `-m` qui permettent d'utiliser la fonctionnalité `use` depuis la ligne de commande.

utime LISTE

Change la date d'accès et de modification de chaque fichier d'une liste de fichiers. Les deux premiers éléments de la liste doivent être, dans l'ordre, les dates NUMÉRIQUES d'accès et de modification. Retourne le nombre de fichiers modifiés avec succès. La date de modification de l'inode de chaque fichier est mis à l'heure courante. Le code suivant a exactement le même effet que la commande `touch` si les fichiers existent déjà :

```
#!/usr/bin/perl
$now = time;
utime $now, $now, @ARGV;
```

Si les deux premiers éléments de LIST valent `undef` alors la fonction `utime(2)` de la bibliothèque C est appelée avec un second argument nul. Sur la plupart des systèmes, cela positionnera les dates d'accès et de modification des fichiers à l'heure courante (comme dans l'exemple ci-dessus).

```
utime undef, undef, @ARGV;
```

values HASH

Retourne la liste de toutes les valeurs contenues dans la table de hachage HASH. (Dans un contexte scalaire, retourne le nombre de valeurs.) Les valeurs sont retournées dans un ordre apparemment aléatoire. Cet ordre peut changer avec les nouvelles versions de Perl mais vous avez la garantie qu'il est identique à celui produit par les fonctions `keys()` ou `each()` appliquées à la même table de hachage (tant qu'elle n'est pas modifiée).

Notez que les valeurs ne sont pas copiées. Ce qui signifie que leur modification entraîne la modification du contenu de la table de hachage :

```
for (values %hash) { s/foo/bar/g } # modifie les valeurs de %hash
for (@hash{keys %hash}) { s/foo/bar/g } # idem
```

L'appel à `values` a pour effet de réinitialiser l'itérateur interne de HASH. Voir aussi [keys](#), [each](#) et [sort](#).

vec EXPR,OFFSET,BITS

Traite la chaîne contenue dans EXPR comme un vecteur de bits dont les éléments sont de la largeur de BITS et retourne un entier non signé contenant la valeur du champ de bits spécifié par OFFSET. BITS spécifie le nombre de bits qui est occupé par chaque entrée dans le vecteur de bits. Cela doit être une puissance de deux entre 1 et 32 (ou 64 si votre plateforme le supporte).

Si BITS vaut 8, les "éléments" coïncident avec les octets de la chaîne d'entrée.

Si BITS vaut 16 ou plus, les octets de la chaîne d'entrée sont groupés par morceau de taille BITS/8 puis chaque groupe est converti en un nombre comme le feraient `pack()` et `unpack()` avec les formats big-endian n/N. Voir [pack](#) pour plus de détails.

Si BITS vaut 4 ou moins, la chaîne est découpée en octets puis les bits de chaque octet sont découpés en 8/BITS groupes. Les bits d'un octet sont numérotés à la manière little-endian comme dans 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80. Par exemple, le découpage d'un seul octet d'entrée `chr(0x36)` en deux groupes donnera la liste (0x6, 0x3) ; son découpage en 4 groupes donnera (0x2, 0x1, 0x3, 0x0).

`vec` peut aussi être affecté auquel cas les parenthèses sont nécessaires pour donner les bonnes priorités :

```
vec($image, $max_x * $x + $y, 8) = 3;
```


Si l'élément sélectionné est en dehors de la chaîne, la valeur retournée sera zéro. Si, au contraire, vous cherchez à écrire dans un élément en dehors de la chaîne, Perl agrandira suffisamment la chaîne au préalable en la complétant par des octets nuls. L'écriture avant le début de la chaîne (c.-à-d avec un OFFSET négatif) est une erreur.

La chaîne ne devrait pas contenir de caractères ayant une valeur > 255 (ceci ne peut arriver que si vous utilisez l'encodage UTF8). Quoiqu'il en soit, elle sera traitée sans tenir compte de son encodage UTF8. Après une affectation via `vec`, les autres parties de votre programme ne considéreront plus cette chaîne comme étant encodée en UTF8. En d'autres termes, si vous avez de tels caractères dans votre chaîne, `vec()` considérera les octets de la chaîne et non les caractères conceptuels qui la composent.

Les chaînes créées par `vec` peuvent aussi être manipulées par les opérateurs logiques `|`, `&` et `^` qui supposent qu'une opération bit à bit est voulue lorsque leurs deux opérandes sont des chaînes. Voir [Opérateurs bit à bit sur les chaînes in the perl op manpage](#).

Le code suivant construit une chaîne ASCII disant 'PerlPerlPerl'. Les commentaires montrent la chaîne après chaque pas. Remarquez que ce code fonctionne de la même manière sur des machines big-endian ou little-endian.

```
my $foo = '';
vec($foo, 0, 32) = 0x5065726C;      # 'Perl'
# $foo eq "Perl" eq "\x50\x65\x72\x6C", 32 bits
print vec($foo, 0, 8);             # affiche 80 == 0x50 == ord('P')
vec($foo, 2, 16) = 0x5065;         # 'PerlPe'
vec($foo, 3, 16) = 0x726C;         # 'PerlPerl'
vec($foo, 8, 8) = 0x50;            # 'PerlPerlP'
vec($foo, 9, 8) = 0x65;           # 'PerlPerlPe'
vec($foo, 20, 4) = 2;              # 'PerlPerlPe' . "\x02"
vec($foo, 21, 4) = 7;              # 'PerlPerlPer'
                                   # 'r' is "\x72"
vec($foo, 45, 2) = 3;              # 'PerlPerlPer' . "\x0c"
vec($foo, 93, 1) = 1;              # 'PerlPerlPer' . "\x2c"
vec($foo, 94, 1) = 1;              # 'PerlPerlPerl'
                                   # 'l' is "\x6c"
```

Pour transformer un vecteur de bits en une chaîne ou en un tableau de 0 et de 1, utilisez ceci :

```
$bits = unpack("b*", $vector);
@bits = split(/,/, unpack("b*", $vector));
```

Si vous connaissez précisément la longueur du vecteur, vous pouvez l'utiliser à la place de `*`.

Voici un exemple qui illustre comment les bits sont réellement placés :

```
#!/usr/bin/perl -wl
print <<'EOT';

                                0          1          2          3
                                unpack("V",$_) 01234567890123456789012345678901
-----
EOT
for $w (0..3) {
    $width = 2**$w;
    for ($shift=0; $shift < $width; ++$shift) {
        for ($off=0; $off < 32/$width; ++$off) {
            $str = pack("B*", "0"x32);
            $bits = (1<<$shift);
            vec($str, $off, $width) = $bits;
            $res = unpack("b*", $str);
            $val = unpack("V", $str);
            write;
        }
    }
}
```

perlfunc – Fonctions Perl prédéfinies

```
format STDOUT =
vec($_, @#, @#) = @<< == @##### @>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
$off, $width, $bits, $val, $res
.
__END__
```

Indépendamment de l'architecture de la machine qui exécute ce code, l'exemple ci-dessus devrait produire la table suivante :

	0	1	2	3
unpack("V", \$_)	01234567890123456789012345678901			
vec(\$_, 0, 1) = 1	==	1	1000000000000000000000000000000000	
vec(\$_, 1, 1) = 1	==	2	0100000000000000000000000000000000	
vec(\$_, 2, 1) = 1	==	4	0010000000000000000000000000000000	
vec(\$_, 3, 1) = 1	==	8	0001000000000000000000000000000000	
vec(\$_, 4, 1) = 1	==	16	0000100000000000000000000000000000	
vec(\$_, 5, 1) = 1	==	32	0000010000000000000000000000000000	
vec(\$_, 6, 1) = 1	==	64	0000001000000000000000000000000000	
vec(\$_, 7, 1) = 1	==	128	0000000100000000000000000000000000	
vec(\$_, 8, 1) = 1	==	256	0000000010000000000000000000000000	
vec(\$_, 9, 1) = 1	==	512	0000000001000000000000000000000000	
vec(\$_, 10, 1) = 1	==	1024	0000000000100000000000000000000000	
vec(\$_, 11, 1) = 1	==	2048	0000000000010000000000000000000000	
vec(\$_, 12, 1) = 1	==	4096	0000000000001000000000000000000000	
vec(\$_, 13, 1) = 1	==	8192	0000000000000100000000000000000000	
vec(\$_, 14, 1) = 1	==	16384	0000000000000010000000000000000000	
vec(\$_, 15, 1) = 1	==	32768	0000000000000001000000000000000000	
vec(\$_, 16, 1) = 1	==	65536	0000000000000000100000000000000000	
vec(\$_, 17, 1) = 1	==	131072	0000000000000000010000000000000000	
vec(\$_, 18, 1) = 1	==	262144	0000000000000000001000000000000000	
vec(\$_, 19, 1) = 1	==	524288	0000000000000000000100000000000000	
vec(\$_, 20, 1) = 1	==	1048576	0000000000000000000010000000000000	
vec(\$_, 21, 1) = 1	==	2097152	0000000000000000000001000000000000	
vec(\$_, 22, 1) = 1	==	4194304	0000000000000000000000100000000000	
vec(\$_, 23, 1) = 1	==	8388608	0000000000000000000000010000000000	
vec(\$_, 24, 1) = 1	==	16777216	0000000000000000000000001000000000	
vec(\$_, 25, 1) = 1	==	33554432	0000000000000000000000000100000000	
vec(\$_, 26, 1) = 1	==	67108864	0000000000000000000000000010000000	
vec(\$_, 27, 1) = 1	==	134217728	0000000000000000000000000001000000	
vec(\$_, 28, 1) = 1	==	268435456	0000000000000000000000000000100000	
vec(\$_, 29, 1) = 1	==	536870912	0000000000000000000000000000010000	
vec(\$_, 30, 1) = 1	==	1073741824	0000000000000000000000000000000100	
vec(\$_, 31, 1) = 1	==	2147483648	0000000000000000000000000000000001	
vec(\$_, 0, 2) = 1	==	1	1000000000000000000000000000000000	
vec(\$_, 1, 2) = 1	==	4	0010000000000000000000000000000000	
vec(\$_, 2, 2) = 1	==	16	0000100000000000000000000000000000	
vec(\$_, 3, 2) = 1	==	64	0000001000000000000000000000000000	
vec(\$_, 4, 2) = 1	==	256	0000000010000000000000000000000000	
vec(\$_, 5, 2) = 1	==	1024	0000000000100000000000000000000000	
vec(\$_, 6, 2) = 1	==	4096	0000000000001000000000000000000000	
vec(\$_, 7, 2) = 1	==	16384	0000000000000010000000000000000000	
vec(\$_, 8, 2) = 1	==	65536	0000000000000000100000000000000000	
vec(\$_, 9, 2) = 1	==	262144	0000000000000000001000000000000000	
vec(\$_, 10, 2) = 1	==	1048576	0000000000000000000010000000000000	
vec(\$_, 11, 2) = 1	==	4194304	0000000000000000000000100000000000	
vec(\$_, 12, 2) = 1	==	16777216	0000000000000000000000001000000000	
vec(\$_, 13, 2) = 1	==	67108864	0000000000000000000000000010000000	
vec(\$_, 14, 2) = 1	==	268435456	0000000000000000000000000000010000	
vec(\$_, 15, 2) = 1	==	1073741824	00000000000000000000000000000000010	
vec(\$_, 0, 2) = 2	==	2	0100000000000000000000000000000000	
vec(\$_, 1, 2) = 2	==	8	0001000000000000000000000000000000	
vec(\$_, 2, 2) = 2	==	32	0000010000000000000000000000000000	

perlfunc – Fonctions Perl prédéfinies

```
vec($_, 0, 8) = 16 ==          16 00001000000000000000000000000000
vec($_, 1, 8) = 16 ==         4096 00000000000001000000000000000000
vec($_, 2, 8) = 16 ==       1048576 000000000000000000000001000000000000
vec($_, 3, 8) = 16 ==     268435456 00000000000000000000000000000001000
vec($_, 0, 8) = 32 ==          32 00000100000000000000000000000000
vec($_, 1, 8) = 32 ==         8192 00000000000001000000000000000000
vec($_, 2, 8) = 32 ==     2097152 000000000000000000000000100000000000
vec($_, 3, 8) = 32 ==   536870912 0000000000000000000000000000000100
vec($_, 0, 8) = 64 ==          64 00000010000000000000000000000000
vec($_, 1, 8) = 64 ==       16384 00000000000000010000000000000000
vec($_, 2, 8) = 64 ==     4194304 0000000000000000000000000100000000
vec($_, 3, 8) = 64 ==  1073741824 0000000000000000000000000000000010
vec($_, 0, 8) = 128 ==         128 00000001000000000000000000000000
vec($_, 1, 8) = 128 ==      32768 0000000000000000000100000000000000
vec($_, 2, 8) = 128 ==   8388608 0000000000000000000000000100000000
vec($_, 3, 8) = 128 == 2147483648 0000000000000000000000000000000001
```

wait

Se comporte comme l'appel système [wait\(2\)](#) sur votre système : attend qu'un processus fils se termine et retourne le pid de ce processus ou `-1` s'il n'y a pas de processus fils. Le statut est retourné par `$?`. Remarquez qu'une valeur de retour `-1` peut signifier que les processus fils ont été automatiquement collectés tel que décrit dans [the perlipc manpage](#).

waitpid PID,FLAGS

Attend qu'un processus fils particulier se termine et retourne le pid de ce processus ou `-1` si ce processus n'existe pas. Sur certains système, une valeur `0` signifie qu'il y a encore des processus actifs. Le statut est retourné par `$?`. Si vous dites :

```
use POSIX ":sys_wait_h";
#...
do {
    $kid = waitpid(-1,&WNOHANG);
} until $kid == -1;
```

alors vous pouvez réaliser une attente non bloquante sur plusieurs processus. Les attentes non bloquantes sont disponibles sur les machines qui connaissent l'un des deux appels système [waitpid\(2\)](#) ou `wait4(2)`. Par contre, l'attente d'un process particulier avec `FLAGS` à `0` est implémenté partout. (Perl émule l'appel système en se souvenant des valeurs du statut des processus qui ont terminé mais qui n'ont pas encore été collectées par le script Perl.)

Remarquez que sur certains systèmes, une valeur de retour `-1` peut signifier que les processus fils ont été automatiquement collectés. Voir [the perlipc manpage](#) pour les détails et d'autres exemples.

wantarray

Retourne `true` (vrai) si le contexte d'appel de la subroutine en cours d'exécution attend une liste de valeurs. Retourne `false` (faux) si le contexte attend un scalaire. Retourne la valeur `undef` si le contexte n'attend aucune valeur (``void context'` ou contexte vide).

```
return unless defined wantarray; # inutile d'en faire plus
my @a = complex_calculation();
return wantarray ? @a : "@a";
```

Cette fonction aurait dû s'appeler `wantlist()`.

warn LISTE

Produit un message d'erreur sur `STDERR` exactement comme [die\(\)](#) mais ne quitte pas et ne génère pas d'exception.

Si LISTE est vide et si \$@ contient encore une valeur (provenant par exemple d'un eval précédent) alors cette valeur est utilisée après y avoir ajouté "\t...caught". C'est pratique pour s'approcher d'un comportement presque similaire à celui de [die\(\)](#).

Si \$@ est vide alors la chaîne "Warning: Something's wrong" (N.d.t: "Attention: quelque chose va mal") est utilisée.

Aucun message n'est affiché si une sous-routine est attachée à \$SIG{__WARN__}. C'est de la responsabilité de cette sous-routine de gérer le message comme elle le veut (en le convertissant en un [die\(\)](#) par exemple). La plupart des sous-routines du genre devraient s'arranger pour afficher réellement les messages qu'elles ne sont pas prêtes à recevoir en appelant à nouveau [warn\(\)](#). Remarquez que cela fonctionne sans produire une boucle sans fin puisque les sous-routines attachées à __WARN__ ne sont pas appelées à partir d'une sous-routine attachée.

Ce comportement est complètement différent de celui des sous-routines attachées à \$SIG{__DIE__} (qui ne peuvent pas supprimer le texte d'erreur mais seulement le remplacer en appelant à nouveau [die\(\)](#)).

L'utilisation d'une sous-routine attachée à __WARN__ fournit un moyen puissant pour supprimer tous les messages d'avertissement (même ceux considérés comme obligatoires).

Un exemple :

```
# supprime *tous* les messages d'avertissement lors de la compilation
BEGIN { $SIG{ '__WARN__' } = sub { warn $_[0] if $DOWARN } }
my $foo = 10;
my $foo = 20;          # pas d'avertissement pour la duplication de
                       # $foo... mais c'est ce qu'on voulait !

# pas de messages d'avertissement avant ici
$DOWARN = 1;
# messages d'avertissement à partir d'ici
warn "\$foo is alive and $foo!";    # devrait apparaître
```

Voir [the perlvar manpage](#) pour plus de détails sur la modification des entrées de %SIG et pour plus d'exemples. Voir le module Carp pour d'autres sortes d'avertissement utilisant les fonctions `carp()` et `cluck()`.

write *DESCRIPTEUR*

write *EXPR*

write

Écrit un enregistrement formaté (éventuellement multi-lignes) vers le DESCRIPTEUR spécifié en utilisant le format associé à ce fichier. Par défaut, le format pour un fichier est celui qui a le même nom que le descripteur mais le format du canal de sortie courant (voir la fonction [select\(\)](#)) peut être spécifié explicitement en stockant le nom du format dans la variable \$~.

Le calcul de l'en-tête est fait automatiquement : si il n'y a pas assez de place sur la page courante pour l'enregistrement formaté, on passe à la page suivante en affichant un format d'en-tête spécial puis on y écrit l'enregistrement formaté. Par défaut, le nom du format d'en-tête spécial est le nom du descripteur auquel on ajoute ``_TOP" mais il peut être dynamiquement modifié en affectant le nom du format voulu à la variable \$^ lorsque le descripteur est sélectionné (par [select\(\)](#)). Le nombre de lignes restant dans la page courante est donné par la variable \$- qui peut être mise à 0 pour forcer le passage à la page suivante.

Si DESCRIPTEUR n'est pas spécifié, la sortie se fait sur le canal de sortie courant qui, au début, est STDOUT mais qui peut être changé par l'opérateur [select\(\)](#). Si le DESCRIPTEUR est une expression EXPR alors l'expression est évaluée et la chaîne résultante est utilisée comme nom du descripteur à utiliser. Pour en savoir plus sur les formats, voir [the perlform manpage](#).

perlfunc – Fonctions Perl prédéfinies

Notez que `write` n'est *PAS* le contraire de `read()`. Malheureusement.

`y///`

L'opérateur de translittération. Identique à `tr///`. Voir [the perlop manpage](#).

3 TRADUCTION

3.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.0. Pour en savoir plus concernant ces traductions, consultez <http://www.enstimac.fr/Perl/>.

3.2 Traducteur

Traduction initiale : Paul Gaborit (paul.gaborit @ enstimac.fr), Jean-Pascal Peltier (jp_peltier @ altavista.net).

Mise à jour en 5.6.0 puis en 5.8.0 : Paul Gaborit (paul.gaborit @ enstimac.fr).

3.3 Relecture

Gérard Delafond. Jean-Louis Morel.