eBook Gratuit

APPRENEZ

Perl Language

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#perl

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: perl-language

It is an unofficial and free Perl Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Perl Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec le langage Perl

Remarques

Perl est le chameau des langues: utile, mais pas toujours beau. Il possède une documentation assez bonne, accessible à l'aide de la commande `perldoc` partir de votre invite de commandes shell. Il est également disponible en ligne sur perldoc.perl.org .

Versions

Version	Notes de version	Date de sortie
1.000		1987-12-18
2.000		1988-06-05
3.000		1989-10-18
4.000		1991-03-21
5.000		1994-10-17
5.001		1995-05-13
5.002		1996-02-29
5.003		1996-06-25
5.004	perl5004delta	1997-05-15
5.005	perl5005delta	1998-07-22
5.6.0	perl56delta	2000-03-22
5.8.0	perl58delta	2002-07-18
5.8.8	perl581delta , perl582delta , perl583delta , perl584delta , perl585delta , perl586delta , perl587delta , perl588delta	2006-02-01
5.10.0	perl5100delta	2007-12-18

Version	Notes de version	Date de sortie
5.12.0	perl5120delta	2010-04-12
5.14.0	Perl5140delta	2011-05-14
5.16.0	perl5160delta	2012-05-20
5.18.0	perl5180delta	2013-05-18
5.20.0	perl5200delta	2014-05-27
5.22.0	perl5220delta	2015-06-01
5.24.0	perl5240delta	2016-05-09
5.26.0	Perl5260delta	2017-05-30

Examples

Démarrer avec Perl

Perl essaie de faire ce que tu veux dire:

```
print "Hello World\n";
```

Les deux bits difficiles sont le point-virgule à la fin de la ligne et le `\n` , qui ajoute une nouvelle ligne (saut de ligne). Si vous avez une version relativement nouvelle de perl, vous pouvez utiliser `say` plutôt que `print` pour que le retour chariot soit ajouté automatiquement:

5.10.0

```
use feature 'say';
say "Hello World";
```

La fonction say est également activée automatiquement avec une `use v5.10` (ou supérieure):

```
use v5.10;
say "Hello World";
```

Il est assez courant d'utiliser perl sur la ligne de commande en utilisant l'option `-e` :

```
$ perl -e 'print "Hello World\n"'
Hello World
```

L'ajout de l'option `-l` est une façon d'imprimer automatiquement les nouvelles lignes:

```
$ perl -le 'print "Hello World"'
Hello World
```

5.10.0

Si vous souhaitez activer de nouvelles fonctionnalités , utilisez plutôt l'option `-E` :

```
$ perl -E 'say "Hello World"'
Hello World
```

Bien entendu, vous pouvez également enregistrer le script dans un fichier. Supprimez simplement l'option de ligne de commande `-e` et utilisez le nom de fichier du script: `perl script.pl` . Pour les programmes plus longs qu'une ligne, il est sage d'activer quelques options:

```
use strict;
use warnings;

print "Hello World\n";
```

Il n'y a pas de désavantage réel autre que de rendre le code légèrement plus long. En échange, le pragma strict vous empêche d'utiliser du code potentiellement dangereux et des avertissements vous avertissent de nombreuses erreurs courantes.

Notez que le point-virgule de fin de ligne est facultatif pour la dernière ligne, mais c'est une bonne idée au cas où vous l'ajouteriez plus tard à la fin de votre code.

Pour plus d'options sur l'exécution de Perl, consultez perlrun ou tapez `perldoc perlrun` à l'invite de commandes. Pour une introduction plus détaillée à Perl, voir perlintro ou tapez `perldoc perlintro` à une invite de commandes. Pour un tutoriel interactif original, essayez Perl .

Lire Démarrer avec le langage Perl en ligne: https://riptutorial.com/fr/perl/topic/341/demarrer-avec-le-langage-perl

Chapitre 2: Analyse XML

Examples

Analyser avec XML :: Twig

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

use strict;
use warnings 'all';

use XML::Twig;

my $twig = XML::Twig->parse( \*DATA );

#we can use the 'root' method to find the root of the XML.
my $root = $twig->root;

#first_child finds the first child element matching a value.
my $title = $root->first_child('title');

#text reads the text of the element.
my $title_text = $title->text;

print "Title is: ", $title_text, "\n";

#The above could be combined:
print $twig ->root->first_child_text('title'), "\n";

## You can use the 'children' method to iterate multiple items:
my $list = $twig->root->first_child('list');

#children can optionally take an element 'tag' - otherwise it just returns all of them.
foreach my $element ( $list->children ) {

   #the 'att' method reads an attribute
   print "Element with ID: ", $element->att('id') // 'none here', " is ", $element->text,
     "\n";
}

#And if we need to do something more complicated, we an use 'xpath'.
#get_xpath or findnodes do the same thing:
#return a list of matches, or if you specify a second numeric argument, just that numbered
match.

#xpath syntax is fairly extensive, but in this one - we search:
# anywhere in the tree: //
#nodes called 'item'
#with an id attribute [@id]
#and with that id attribute equal to "1000".
#by specifying '0' we say 'return just the first match'.

print "Item 1000 is: ", $twig->get_xpath( '//item[@id="1000"]', 0 )->text, "\n";

#this combines quite well with `map` to e.g. do the same thing on multiple items
print "All IDs:\n", join ( "\n", map { $_ -> att('id') } $twig -> get_xpath('//item'));
#note how this also finds the item under 'summary', because of //
```

```
__DATA__
<?xml version="1.0" encoding="utf-8"?>
<root>
  <title>some sample xml</title>
  <first key="value" key2="value2">
    <second>Some text</second>
  </first>
  <third>
    <fourth key3="value">Text here too</fourth>
  </third>
  <list>
     <item id="1">Item1</item>
     <item id="2">Item2</item>
     <item id="3">Item3</item>
     <item id="66">Item66</item>
     <item id="88">Item88</item>
     <item id="100">Item100</item>
     <item id="1000">Item1000</item>
     <notanitem>Not an item at all really.</notanitem>
  </list>
  <summary>
     <item id="no_id">Test</item>
  </summary>
</root>
```

Consommer XML avec XML :: Rabbit

Avec `XML::Rabbit` il est possible de consommer facilement des fichiers XML. Vous *définissez* de manière déclarative et avec une syntaxe XPath ce que vous recherchez dans XML et `XML::Rabbit` renverra des objets selon la définition donnée.

Définition:

```
package Bookstore;
use XML::Rabbit::Root;
has_xpath_object_list books => './book' => 'Bookstore::Book';
finalize_class();

package Bookstore::Book;
use XML::Rabbit;
has_xpath_value bookid => './@id';
has_xpath_value author => './author';
has_xpath_value title => './title';
has_xpath_value genre => './genre';
has_xpath_value price => './price';
has_xpath_value publish_date => './publish_date';
has_xpath_value description => './description';
has_xpath_object purchase_data => './purchase_data' => 'Bookstore::Purchase';
finalize_class();

package Bookstore::Purchase;
use XML::Rabbit;
has_xpath_value price => './price';
has_xpath_value date => './date';
finalize_class();
```

Consommation XML:

```
use strict;
use warnings;
use utf8;

package Library;
use feature qw(say);
use Carp;
use autodie;

say "Showing data information";
my $bookstore = Bookstore->new( file => './sample.xml' );

foreach my $book( @{$bookstore->books} ) {
    say "ID: " . $book->bookid;
    say "Title: " . $book->title;
    say "Author: " . $book->author, "\n";
}
```

Remarques:

S'il vous plaît soyez prudent avec les éléments suivants:

1. Le premier cours doit être `XML::Rabbit::Root` . Il vous placera dans la balise principale du document XML. Dans notre cas, cela nous placera dans `<catalog>`

2. Les classes imbriquées sont facultatives. Ces classes doivent être accessibles via un bloc try / catch (ou `eval / $@` check). Les champs facultatifs renverront simplement `null` . Par exemple, pour `purchase_data` la boucle serait:

```
foreach my $book( @{$bookstore->books} ) {
    say "ID: " . $book->bookid;
    say "Title: " . $book->title;
    say "Author: " . $book->author;
    try {
        say "Purchase price: ". $book->purchase_data->price, "\n";
    } catch {
        say "No purchase price available\n";
    }
}
```

sample.xml

```
<?xml version="1.0"?>
<catalog>
   <book id="bk101">
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>44.95</price>
      <publish_date>2000-10-01</publish_date>
      <description>An in-depth look at creating applications
      with XML.</description>
   </book>
   <book id="bk102">
```

```
      <author>Ralls, Kim</author>
      <title>Midnight Rain</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2000-12-16</publish_date>
      <description>A former architect battles corporate zombies,
      an evil sorceress, and her own childhood to become queen
      of the world.</description>
   </book>
   <book id="bk103">
      <author>Corets, Eva</author>
      <title>Maeve Ascendant</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2000-11-17</publish_date>
      <description>After the collapse of a nanotechnology
      society in England, the young survivors lay the
      foundation for a new society.</description>
   </book>
   <book id="bk104">
      <author>Corets, Eva</author>
      <title>Oberon's Legacy</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2001-03-10</publish_date>
      <description>In post-apocalypse England, the mysterious
      agent known only as Oberon helps to create a new life
      for the inhabitants of London. Sequel to Maeve
      Ascendant.</description>
      <purchase_data>
        <date>2001-12-21</date>
        <price>20</price>
      </purchase_data>
   </book>
</catalog>
```

Analyse avec XML :: LibXML

```
# This uses the 'sample.xml' given in the XML::Twig example.

# Module requirements (1.70 and above for use of load_xml)
use XML::LibXML '1.70';

# let's be a good perl dev
use strict;
use warnings 'all';

# Create the LibXML Document Object
my $xml = XML::LibXML->new();

# Where we are retrieving the XML from
my $file = 'sample.xml';

# Load the XML from the file
my $dom = XML::LibXML->load_xml(
    location => $file
);

# get the docroot
```

```
my $root = $dom->getDocumentElement;

# if the document has children
if($root->hasChildNodes) {

    # getElementsByLocalName returns a node list of all elements who's
    # localname matches 'title', and we want the first occurrence
    # (via get_node(1))
    my $title = $root->getElementsByLocalName('title');

    if(defined $title) {
        # Get the first matched node out of the nodeList
        my $node = $title->get_node(1);

        # Get the text of the target node
        my $title_text = $node->textContent;

        print "The first node with name 'title' contains: $title_text\n";
    }

    # The above calls can be combined, but is possibly prone to errors
    # (if the getElementsByLocalName() failed to match a node).
    #
    # my $title_text = $root->getElementsByLocalName('title')->get_node(1)->textContent;
}

# Using Xpath, get the price of the book with id 'bk104'
#

# Set our xpath
my $xpath = q!/catalog/book[@id='bk104']/price!;

# Does that xpath exist?
if($root->exists($xpath)) {

    # Pull in the twig
    my $match = $root->find($xpath);

    if(defined $match) {
        # Get the first matched node out of the nodeList
        my $node = $match->get_node(1);

        # pull in the text of that node
        my $match_text = $node->textContent;

        print "The price of the book with id bk104 is: $match_text\n";
    }
}
```

Lire Analyse XML en ligne: https://riptutorial.com/fr/perl/topic/3590/analyse-xml

Chapitre 3: Applications GUI en Perl

Remarques

Tk est l'un des outils d'interface graphique les plus utilisés pour Perl. Les autres outils les plus courants sont les widgets GTK + 2 et 3, WxWidgets et Win32. Qt4, XUL, Prima et FLTK sont moins utilisés.

Tk, GTK + 3, Wx, Win32, Prima, FLTK et XUL sont activement mis à jour. Qt4 et GTK + 2 ne sont plus développés activement, mais peuvent avoir des versions de maintenance.

Examples

Application GTK

```
use strict;
use warnings;

use Gtk2 -init;

my $window = Gtk2::Window->new();
$window->show();

Gtk2->main();

0;
```

Lire Applications GUI en Perl en ligne: https://riptutorial.com/fr/perl/topic/5924/applications-gui-en-perl

Chapitre 4: Au hasard

Remarques

Documentation pour la fonction rand (): http://perldoc.perl.org/functions/rand.html

Examples

Générer un nombre aléatoire entre 0 et 100

Passer une limite supérieure en tant qu'argument à la fonction rand ().

Contribution:

```
my $upper_limit = 100;
my $random = rand($upper_limit);

print $random . "\n";
```

Sortie:

Un nombre à virgule flottante aléatoire, comme ...

```
45.8733038119139
```

Générer un entier aléatoire entre 0 et 9

Lancez votre nombre à virgule flottante aléatoire comme un int.

Contribution:

```
my $range = 10;

# create random integer as low as 0 and as high as 9
my $random = int(rand($range));   # max value is up to but not equal to $range

print $random . "\n";
```

Sortie:

Un entier aléatoire, comme ...

```
0
```

Voir aussi le perldoc pour le rand .

Accéder à un élément de tableau au hasard

```
my @letters = ( 'a' .. 'z' );                 # English ascii-bet

print $letters[ rand @letters ] for 1 .. 5;  # prints 5 letters at random
```

Comment ça marche

- `rand EXPR` attend une valeur scalaire, donc `@letters` est évalué dans un contexte scalaire
- Un tableau dans un contexte scalaire renvoie le nombre d'éléments qu'il contient (26 dans ce cas)
- `rand 26` renvoie un nombre fractionnaire aléatoire dans l'intervalle `0 ≤ VALUE < 26` . (Ça ne peut jamais être `26`)
- Les index de tableau sont toujours des entiers, donc `$letters[rand @letters]` `$letters[int rand @letters]`
- Les tableaux Perl sont indexés à zéro, donc `$array[rand @array]` renvoie `$array[0]` , `$array[$#array]` ou un élément entre

---

(Le même principe s'applique aux hachages)

```
my %colors = ( red   => 0xFF0000,
               green => 0x00FF00,
               blue  => 0x0000FF,
             );

print ( values %colors )[rand keys %colors];
```

Lire Au hasard en ligne: https://riptutorial.com/fr/perl/topic/6905/au-hasard

Chapitre 5: Chaînes et méthodes de citation

Remarques

La syntaxe de la version ne nous permet pas de protéger les versions qui n'existent pas encore, donc ceci est un rappel pour que quelqu'un les modifie et les édite une fois qu'il arrive (RE: Perl 5.26). Les gardiens de version doivent plutôt avoir une classification «future» des fonctions provisoires pouvant être disponibles pour les personnes assez courageuses pour effectuer une vérification de la source.

Examples

Citation littérale de chaîne

Les littéraux de chaîne n'impliquent aucune fuite ou interpolation (à l'exception des guillemets de fin de chaîne)

```
print 'This is a string literal\n'; # emits a literal \ and n to terminal

print 'This literal contains a \'postraphe '; # emits the ' but not its preceding \
```

Vous pouvez utiliser d'autres mécanismes de citation pour éviter les conflits:

```
print q/This is is a literal \' <-- 2 characters /;  # prints both \ and '
print q^This is is a literal \' <-- 2 characters ^;  # also
```

Certains caractères de citation choisis sont "équilibrés"

```
print q{ This is a literal and I contain { parens! } }; # prints inner { }
```

Double cotation

Les chaînes entre guillemets doubles utilisent l' **interpolation** et l' **échappement** , contrairement aux chaînes entre guillemets simples. Pour citer une chaîne entre guillemets, utilisez des guillemets doubles `"` ou l'opérateur `qq` .

```
my $greeting = "Hello!\n";
print $greeting;
# => Hello! (followed by a linefeed)

my $bush = "They misunderestimated me."
print qq/As Bush once said: "$bush"\n/;
# => As Bush once said: "They misunderestimated me." (with linefeed)
```

Le `qq` est utile ici pour éviter de devoir échapper aux guillemets. Sans elle, il faudrait écrire ...

```
print "As Bush once said: \"$bush\"\n";
```

... ce qui n'est pas si beau

Perl ne vous limite pas à utiliser une barre oblique `/` avec `qq` ; vous pouvez utiliser n'importe quel caractère (visible).

```
use feature 'say';

say qq/You can use slashes.../;
say qq{...or braces...};
say qq^...or hats...^;
say qq|...or pipes...|;
# say qq ...but not whitespace. ;
```

Vous pouvez également interpoler des tableaux en chaînes.

```
use feature 'say';

my @letters = ('a', 'b', 'c');
say "I like these letters: @letters.";
# => I like these letters: a b c.
```

Par défaut, les valeurs sont séparées par des espaces - car la variable spéciale `$"` utilise par défaut un seul espace. Cela peut bien sûr être modifié.

```
use feature 'say';

my @letters = ('a', 'b', 'c');
{local $" = ", "; say "@letters"; }    # a, b, c
```

Si vous préférez, vous avez la possibilité d' `use English` et de modifier `$LIST_SEPARATOR` place:

```
use v5.18; # English should be avoided on older Perls
use English;

my @letters = ('a', 'b', 'c');
{ local $LIST_SEPARATOR = "\n"; say "My favourite letters:\n\n@letters" }
```

Pour quelque chose de plus complexe que cela, vous devriez utiliser une boucle à la place.

```
say "My favourite letters:";
say;
for my $letter (@letters) {
  say " - $letter";
}
```

L'interpolation ne fonctionne *pas* avec les hachages.

```
use feature 'say';

my %hash = ('a', 'b', 'c', 'd');
```

```
say "This doesn't work: %hash"         # This doesn't work: %hash
```

Certains codes abusent de l'interpolation des références - **évitez-le** .

```
use feature 'say';

say "2 + 2 == @{[ 2 + 2 ]}";           # 2 + 2 = 4 (avoid this)
say "2 + 2 == ${\( 2 + 2 )}";          # 2 + 2 = 4 (avoid this)
```

Le soi-disant "opérateur de panier" amène perl à déréférencer `@{ ... }` la référence de tableau qui `[ ... ]` contient l'expression que vous voulez interpoler, `2 + 2` . Lorsque vous utilisez cette astuce, Perl construit un tableau anonyme, puis le déréférencera et le supprimera.

La version `${\( ... )}` est un peu moins coûteuse, mais il faut quand même allouer de la mémoire et la lecture est encore plus difficile.

Au lieu de cela, envisagez d'écrire:

- `say "2 + 2 == " . 2 + 2;`
- `my $result = 2 + 2; say "2 + 2 == $result"`

Heredocs

Les grandes chaînes multi-lignes sont lourdes à écrire.

```
my $variable = <<'EOF';
this block of text is interpreted literally,
no \'quotes matter, they're just text
only the trailing left-aligned EOF matters.
EOF
```

NB: Assurez-vous d'ignorer le surligneur de syntaxe de dépassement de pile: c'est très faux.

Et Interpolated Heredocs fonctionne de la même manière.

```
my $variable = <<"I Want it to End";
this block of text is interpreted.
quotes\nare interpreted, and $interpolations
get interpolated...
but still, left-aligned "I Want it to End" matters.
I Want it to End
```

En attente dans 5.26.0 * est une syntaxe "Indented Heredoc" qui coupe le remplissage à gauche pour vous

5.26.0

```
my $variable = <<~"MuchNicer";
    this block of text is interpreted.
    quotes\nare interpreted, and $interpolations
    get interpolated...
```

```
    but still, left-aligned "I Want it to End" matters.
MuchNicer
```

Supprimer les nouvelles lignes

La fonction `chomp` supprime *un* caractère de nouvelle ligne, s'il est présent, de chaque scalaire qui lui est transmis. `chomp` mutera la chaîne d'origine et retournera le nombre de caractères supprimés

```
my $str = "Hello World\n\n";
my $removed = chomp($str);
print $str;     # "Hello World\n"
print $removed; # 1

# chomp again, removing another newline
$removed = chomp $str;
print $str;     # "Hello World"
print $removed; # 1

# chomp again, but no newline to remove
$removed = chomp $str;
print $str;     # "Hello World"
print $removed; # 0
```

Vous pouvez également `chomp` plusieurs chaînes à la fois:

```
my @strs = ("Hello\n", "World!\n\n"); # one newline in first string, two in second

my $removed = chomp(@strs); # @strs is now  ("Hello", "World!\n")
print $removed;             # 2

$removed = chomp(@strs); # @strs is now ("Hello", "World!")
print $removed;          # 1

$removed = chomp(@strs); # @strs is still ("Hello", "World!")
print $removed;          # 0
```

Mais généralement, personne ne s'inquiète du nombre de lignes supprimées, donc `chomp` est généralement vu dans un contexte vide et généralement dû à la lecture des lignes d'un fichier:

```
while (my $line = readline $fh)
{
    chomp $line;

    # now do something with $line
}

my @lines = readline $fh2;

chomp (@lines); # remove newline from end of each line
```

Lire Chaînes et méthodes de citation en ligne: https://riptutorial.com/fr/perl/topic/1984/chaines-et-methodes-de-citation

Chapitre 6: Commandes Perl pour Windows Excel avec le module Win32 :: OLE

Introduction

Ces exemples présentent les commandes les plus utilisées de Perl pour manipuler Excel via le module Win32 :: OLE.

Syntaxe

- $ Sheet-> Range (*Cell1* , [*Cell2*]) #Sélectionnez une cellule ou une plage de cellules
- $> Cellules en forme de feuille *(rowIndex, columnIndex)* #select une cellule par l' indice de rangée et de colonne

Paramètres

Paramètres	Détails
Cell1 (requis)	Le nom de la plage. Ce doit être une référence de style A1 dans la langue de la macro. Il peut inclure l'opérateur d'intervalle (un deux-points), l'opérateur d'intersection (un espace) ou l'opérateur d'union (une virgule).
Cell2 (optionnel)	Si spécifié, *Cell1* correspond au coin supérieur gauche de la plage et *Cell2* correspond au coin inférieur droit de la plage.

Remarques

Lien pour des informations sur les couleurs sur Excel: http://dmcritchie.mvps.org/excel/colors.htm

	A	B	C	D
1		1		29
2		2		30
3		3		31
4		4		32
5		5		33
6		6		34
7		7		35
8		8		36
9		9		37
10		10		38
11		11		39
12		12		40
13		13		41
14		14		42
15		15		43
16		16		44
17		17		45
18		18		46
19		19		47
20		20		48
21		21		49
22		22		50
23		23		51
24		24		52
25		25		53
26		26		54
27		27		55
28		28		56

Lien pour des informations sur les constantes Excel: http://msdn.microsoft.com/en-us/library/aa221100%28office.11%29.aspx

Liens depuis le module Win32 :: OLE: http://search.cpan.org/~jdb/Win32-OLE-0.1712/lib/Win32/OLE.pm#EXAMPLES

Des informations utiles sur l'utilisation d'Excel peuvent être trouvées à cette adresse

Examples

1. Ouverture et enregistrement de classeurs Excel

```
#Modules to use
use Cwd 'abs_path';
use Win32::OLE;
use Win32::OLE qw(in with);
use Win32::OLE::Const "Microsoft Excel";
$Win32::OLE::Warn = 3;

#Need to use absolute path for Excel files
my $excel_file = abs_path("$Excel_path") or die "Error: the file $Excel_path has not been
found\n";

# Open Excel application
my $Excel = Win32::OLE->GetActiveObject('Excel.Application')
```

```
    || Win32::OLE->new('Excel.Application', 'Quit');

# Open Excel file
my $Book = $Excel->Workbooks->Open($excel_file);

#Make Excel visible
$Excel->{Visible} = 1;

#___ ADD NEW WORKBOOK
my $Book = $Excel->Workbooks->Add;
my $Sheet = $Book->Worksheets("Sheet1");
$Sheet->Activate;

#Save Excel file
$Excel->{DisplayAlerts}=0; # This turns off the "This file already exists" message.
$Book->Save; #Or $Book->SaveAs("C:\\file_name.xls");
$Book->Close; #or $Excel->Quit;
```

2. Manipulation des feuilles de travail

```
#Get the active Worksheet
my $Book = $Excel->Activewindow;
my $Sheet = $Book->Activesheet;

#List of Worksheet names
my @list_Sheet = map { $_->{'Name'} } (in $Book->{Worksheets});

#Access a given Worksheet
my $Sheet = $Book->Worksheets($list_Sheet[0]);

#Add new Worksheet
$Book->Worksheets->Add({After => $workbook->Worksheets($workbook->Worksheets->{Count})});

#Change Worksheet Name
$Sheet->{Name} = "Name of Worksheet";

#Freeze Pane
$Excel -> ActiveWindow -> {FreezePanes} = "True";

#Delete Sheet
$Sheet -> Delete;
```

3. Manipulation des cellules

```
#Edit the value of a cell (2 methods)
$Sheet->Range("A1")->{Value} = 1234;
$Sheet->Cells(1,1)->{Value} = 1234;

#Edit the values in a range of cells
$Sheet->Range("A8:C9")->{Value} = [[ undef, 'Xyzzy', 'Plugh' ],
                                  [ 42,    'Perl',  3.1415  ]];

#Edit the formula in a cell (2 types)
$Sheet->Range("A1")->{Formula} = "=A1*9.81";
$Sheet->Range("A3")->{FormulaR1C1} = "=SUM(R[-2]C:R[-1]C)";      # Sum of rows
$Sheet->Range("C1")->{FormulaR1C1} = "=SUM(RC[-2]:RC[-1])";      # Sum of columns
```

```
#Edit the format of the text (font)
$Sheet->Range("G7:H7")->Font->{Bold}       = "True";
$Sheet->Range("G7:H7")->Font->{Italic}     = "True";
$Sheet->Range("G7:H7")->Font->{Underline}  = xlUnderlineStyleSingle;
$Sheet->Range("G7:H7")->Font->{Size}       = 8;
$Sheet->Range("G7:H7")->Font->{Name}       = "Arial";
$Sheet->Range("G7:H7")->Font->{ColorIndex} = 4;

#Edit the number format
$Sheet -> Range("G7:H7") -> {NumberFormat} = "\@";                                  # Text
$Sheet -> Range("A1:H7") -> {NumberFormat} = "\$#,##0.00";                         # Currency
$Sheet -> Range("G7:H7") -> {NumberFormat} = "\$#,##0.00_);[Red](\$#,##0.00)";      # Currency
- red negatives
$Sheet -> Range("G7:H7") -> {NumberFormat} = "0.00_);[Red](0.00)";                  # Numbers
with decimals
$Sheet -> Range("G7:H7") -> {NumberFormat} = "#,##0";                           #     Numbers
with commas
$Sheet -> Range("G7:H7") -> {NumberFormat} = "#,##0_);[Red](#,##0)";                # Numbers
with commas - red negatives
$Sheet -> Range("G7:H7") -> {NumberFormat} = "0.00%";                               # Percents
$Sheet -> Range("G7:H7") -> {NumberFormat} = "m/d/yyyy"                             # Dates

#Align text
$Sheet -> Range("G7:H7") -> {HorizontalAlignment} = xlHAlignCenter;                 # Center
text;
$Sheet -> Range("A1:A2") -> {Orientation} = 90;                                     # Rotate
text

#Activate Cell
$Sheet -> Range("A2") -> Activate;

$Sheet->Hyperlinks->Add({
   Anchor          =>  $range, #Range of cells with the hyperlink; e.g. $Sheet->Range("A1")
   Address         =>  $adr, #File path, http address, etc.
   TextToDisplay   =>  $txt, #Text in the cell
   ScreenTip       =>  $tip, #Tip while hovering the mouse over the hyperlink
});
```

NB: pour récupérer la liste des liens hypertextes, consultez l'article suivant Obtention de la liste des liens hypertexte d'une feuille de calcul Excel avec Perl Win32 :: OLE

4. Manipulation des lignes / colonnes

```
#Insert a row before/after line 22
$Sheet->Rows("22:22")->Insert(xlUp, xlFormatFromRightOrBelow);
$Sheet->Rows("23:23")->Insert(-4121,0);     #xlDown is -4121 and that xlFormatFromLeftOrAbove
is 0

#Delete a row
$Sheet->Rows("22:22")->Delete();

#Set column width and row height
$Sheet -> Range('A:A') -> {ColumnWidth} = 9.14;
$Sheet -> Range("8:8") -> {RowHeight}   = 30;
$Sheet -> Range("G:H") -> {Columns} -> Autofit;

# Get the last row/column
my $last_row = $Sheet -> UsedRange -> Find({What => "*", SearchDirection => xlPrevious,
SearchOrder => xlByRows})    -> {Row};
```

```
my $last_col = $Sheet -> UsedRange -> Find({What => "*", SearchDirection => xlPrevious,
SearchOrder => xlByColumns}) -> {Column};


#Add borders (method 1)
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {LineStyle}  = xlDouble;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {Weight}     = xlThick;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {ColorIndex} = 1;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeLeft)         -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeLeft)         -> {Weight}     = xlThin;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeTop)          -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeTop)          -> {Weight}     = xlThin;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {Weight}     = xlThin;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeRight)        -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeRight)        -> {Weight}     = xlThin;
$Sheet -> Range("A3:H3") -> Borders(xlInsideVertical)   -> {LineStyle}  = xlDashDot
$Sheet -> Range("A3:H3") -> Borders(xlInsideVertical)   -> {Weight}     = xlMedium;
$Sheet -> Range("A3:I3") -> Borders(xlInsideHorizontal) -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:I3") -> Borders(xlInsideHorizontal) -> {Weight}     = xlThin;

#Add borders (method 2)
my @edges = qw (xlInsideHorizontal xlInsideVertical xlEdgeBottom xlEdgeTop xlEdgeRight);
foreach my $edge (@edges)
```

Lire Commandes Perl pour Windows Excel avec le module Win32 :: OLE en ligne: https://riptutorial.com/fr/perl/topic/3420/commandes-perl-pour-windows-excel-avec-le-module-win32----ole

Chapitre 7: commentaires

Examples

Commentaires sur une seule ligne

Les commentaires sur une seule ligne commencent par un signe dièse `#` et vont à la fin de la ligne:

```
# This is a comment

my $foo = "bar"; # This is also a comment
```

Commentaires multilignes

Les commentaires sur plusieurs lignes commencent par `=` et avec l'instruction `=cut` . Ce sont des commentaires spéciaux appelés POD (Plain Old Documentation).

Tout texte entre les marqueurs sera commenté:

```
=begin comment

This is another comment.
And it spans multiple lines!

=end comment

=cut
```

Lire commentaires en ligne: https://riptutorial.com/fr/perl/topic/6815/commentaires

Chapitre 8: Compiler le module cpan Perl sapnwrfc à partir du code source

Introduction

J'aimerais décrire les conditions préalables et les étapes à suivre pour construire le module perl CPAN sapnwrfc avec l'environnement Strawberry Perl sous Windows 7 x64. Il devrait également fonctionner pour toutes les versions ultérieures de Windows telles que 8, 8.1 et 10.

J'utilise Strawberry Perl 5.24.1.1 64 bit mais il devrait aussi fonctionner avec les anciennes versions.

Il m'a fallu un peu de temps pour réussir avec plusieurs tentatives (installation de 32 vs 64 bits de Perl, SDK RFC SDK, compilateur MinGW vs Microsoft C). J'espère donc que certains bénéficieront de mes découvertes.

Remarques

Installez un package Strawberry Perl 64 bits à partir de http://strawberryperl.com . Dans mon cas, c'était 5.24.1.1.

Téléchargez la version actuelle du bit SDK RFC SDK x64 à partir de https://launchpad.support.sap.com/#/softwarecenter

Vous pouvez le trouver avec la trace suivante: **Support Packages et Patches => Par catégorie => Composants supplémentaires => SDK RFC SAP NW => SDK RFC SDK 7.20**

Dans mon cas, la version actuelle était 7.20 PL42 x64.

Extrayez le fichier téléchargé avec `sapcar -xvf NWRFC_42-20004568.SAR`

J'ai renommé le dossier en `C:\nwrfcsdk_x64`

Créez des fichiers .def et .a pour le compilateur / éditeur de liens MinGW avec les commandes suivantes dans le répertoire C: \ nwrfcsdk_x64:

```
gendef *.dll
dlltool --dllname icuin34.dll --def icuin34.def --output-lib icuin34.a
dlltool --dllname icudt34.dll --def icudt34.def --output-lib icudt34.a
dlltool --dllname icuuc34.dll --def icuuc34.def --output-lib icuuc34.a
dlltool --dllname libsapucum.dll --def libsapucum.def --output-lib libsapucum.a
dlltool --dllname libicudecnumber.dll --def libicudecnumber.def --output-lib libicudecnumber.a
dlltool --dllname sapnwrfc.dll --def sapnwrfc.def --output-lib sapnwrfc.a
```

Dans le répertoire C: \ nwrfcsdk_x64 \ lib, les fichiers suivants doivent exister:

```
icudt34.a
```

```
icudt34.def
icudt34.dll
icuin34.a
icuin34.def
icuin34.dll
icuuc34.a
icuuc34.def
icuuc34.dll
libicudecnumber.a
libicudecnumber.def
libicudecnumber.dll
libsapucum.a
libsapucum.def
libsapucum.dll
libsapucum.lib
sapdecfICUlib.lib
sapnwrfc.a
sapnwrfc.def
sapnwrfc.dll
sapnwrfc.lib
```

Démarrez l'invite de commande avec `cmd.exe` et démarrez le programme `cpan` .

Lancez la commande `get sapnwrfc` pour télécharger le module Perl sapnwrfc depuis CPAN.

Laissez l'environnement cpan avec la commande `exit` . Changez de répertoire en `C:\Strawberry\cpan\build\sapnwrfc-0.37-0` .

Construisez le ou les fichiers Make avec la commande suivante. Adaptez les noms de dossier en fonction de votre configuration.

```
perl Makefile.PL --source=C:\nwrfcsdk_x64 --addlibs "C:\nwrfcsdk_x64\lib\sapnwrfc.a
C:\nwrfcsdk_x64\lib\libsapucum.a"
```

Exécutez les commandes `dmake` et `dmake install` pour créer et installer le module.

Copiez les fichiers de `C:\nwrfcsdk_x64\lib` vers `C:\Strawberry\perl\site\lib\auto\SAPNW\Connection` .

Examples

Exemple simple pour tester la connexion RFC

Exemple simple à partir de http://search.cpan.org/dist/sapnwrfc/sapnwrfc-cookbook.pod

```
use strict;
use warnings;
use utf8;
use sapnwrfc;

SAPNW::Rfc->load_config('sap.yml');
my $conn = SAPNW::Rfc->rfc_connect;

my $rd = $conn->function_lookup("RPY_PROGRAM_READ");
my $rc = $rd->create_function_call;
```

```
$rc->PROGRAM_NAME("SAPLGRFC");

eval {
$rc->invoke;
};
if ($@) {
    die "RFC Error: $@\n";
}

print "Program name: ".$rc->PROG_INF->{'PROGNAME'}."\n";
my $cnt_lines_with_text = scalar grep(/LGRFCUXX/, map { $_->{LINE} } @{$rc->SOURCE_EXTENDED});
$conn->disconnect;
```

Lire Compiler le module cpan Perl sapnwrfc à partir du code source en ligne: https://riptutorial.com/fr/perl/topic/9775/compiler-le-module-cpan-perl-sapnwrfc-a-partir-du-code-source

Chapitre 9: Danseur

Introduction

Sur:

Dancer2 (le successeur de Dancer) est un framework d'application web simple mais puissant pour Perl.

Il est inspiré par Sinatra et écrit par Alexis Sukrieh.

Principales caractéristiques: ••• Dead Simple - Syntaxe intuitive, minimaliste et très expressive. ••• Flexible - Le support PSGI, les plug-ins et la conception modulaire permettent une forte évolutivité. ••• Peu de dépendances - Dancer dépend le moins possible de modules CPAN, ce qui facilite son installation.

Examples

Exemple le plus simple

```
#!/usr/bin/env perl
use Dancer2;

get '/' => sub {
    "Hello World!"
};

dance;
```

Lire Danseur en ligne: https://riptutorial.com/fr/perl/topic/5921/danseur

Chapitre 10: Dates et heure

Examples

Créer un nouveau date

Installez `DateTime` sur votre PC, puis utilisez-le en script perl:

```
use DateTime;
```

Créer un nouveau datetime

```
$dt = DateTime->now( time_zone => 'Asia/Ho_Chi_Minh');
```

Vous pouvez ensuite accéder aux valeurs de date et d'heure des éléments:

```
$year   = $dt->year;
$month  = $dt->month;
$day    = $dt->day;
$hour   = $dt->hour;
$minute = $dt->minute;
$second = $dt->second;
```

Pour avoir seulement du temps:

`my $time = $dt->hms;` #retour time au format `hh:mm:ss`

Pour obtenir uniquement la date:

`my $date = $dt->ymd;` date de retour au format `yyyy-mm-dd`

Travailler avec des éléments de datetime

Définir un seul élément:

```
$dt->set( year => 2016 );
```

Définir de nombreux éléments:

```
$dt->set( year => 2016, 'month' => 8);
```

Ajouter une durée à datetime

```
$dt->add( hour => 1, month => 2)
```

Soustraction de datetime:

```
my $dt1 = DateTime->new(
      year       => 2016,
      month      => 8,
      day        => 20,
```

```
);

my $dt2 = DateTime->new(
      year       => 2016,
      month      => 8,
      day        => 24,
);

my $duration = $dt2->subtract_datetime($dt1);
print $duration->days
```

Vous obtiendrez le résultat est 4 jours

Calculer le temps d'exécution du code

```
use Time::HiRes qw( time );

my $start = time();

#Code for which execution time is calculated
sleep(1.2);

my $end = time();

printf("Execution Time: %0.02f s\n", $end - $start);
```

Cela imprimera le temps d'exécution du code en secondes

Lire Dates et heure en ligne: https://riptutorial.com/fr/perl/topic/5920/dates-et-heure

Chapitre 11: Dates et heure

Examples

Formatage de la date

Time :: Piece est disponible en Perl 5 après la version 10

```
use Time::Piece;

my $date = localtime->strftime('%m/%d/%Y');
print $date;
```

```
Output
07/26/2016
```

Lire Dates et heure en ligne: https://riptutorial.com/fr/perl/topic/5923/dates-et-heure

Chapitre 12: Débogage du script Perl

Examples

Exécuter le script en mode débogage

Pour exécuter le script en mode débogage, vous devez ajouter l'option `-d` dans la ligne de commande:

```
$perl -d script.pl
```

Si `t` est spécifié, cela indique au débogueur que les threads seront utilisés dans le code en cours de débogage:

```
$perl -dt script.pl
```

Informations complémentaires sur `perldoc` perlrun

Utilisez un débogueur non standard

`$perl -d:MOD script.pl` exécute le programme sous le contrôle d'un module de débogage, de profilage ou de traçage installé en tant que `Devel::MOD` .

Par exemple, `-d:NYTProf` exécute le programme à l'aide du profileur `Devel::NYTProf` .

Voir tous les modules Devel disponibles ici

Modules recommandés:

- `Devel::NYTProf` - Puissant profileur de code source Perl riche en fonctionnalités
- `Devel::Trepan` - Un débogueur Perl modulaire de type gdb
- `Devel::MAT` - Outil d'analyse de mémoire Perl
- `Devel::hdb` - `Devel::hdb` Perl en tant que page Web et service REST
- `Devel::DebugHooks::KillPrint` - Permet d'oublier le débogage par `print`
- `Devel::REPL` - Un shell interactif perl moderne
- `Devel::Cover` - `Devel::Cover` code pour Perl

Lire Débogage du script Perl en ligne: https://riptutorial.com/fr/perl/topic/6769/debogage-du-script-perl

Chapitre 13: Déboguer la sortie

Examples

Dumping data-structures

```
use Data::Dumper;

my $data_structure = { foo => 'bar' };
print Dumper $data_structure;
```

L'utilisation de Data :: Dumper est un moyen simple d'examiner les structures de données ou le contenu variable au moment de l'exécution. Il est livré avec Perl et vous pouvez le charger facilement. La fonction `Dumper` renvoie la structure de données sérialisée d'une manière qui ressemble à du code Perl.

```
$VAR1 = {
        'foo' => 'bar',
}
```

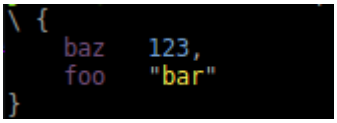
Cela rend très utile d'examiner rapidement certaines valeurs de votre code. C'est l'un des outils les plus pratiques de votre arsenal. Lisez la documentation complète sur metacpan .

Dumping avec style

Parfois, Data :: Dumper ne suffit pas. Vous avez un objet orignal que vous voulez inspecter? Des nombres énormes de la même structure? Vous voulez des choses triées? Coloré? Data :: Printer est votre ami.

```
use Data::Printer;

p $data_structure;
```



Data :: Printer écrit sur STDERR, comme `warn` . Cela facilite la recherche de la sortie. Par défaut, il trie les clés de hachage et examine les objets.

```
use Data::Printer;
use LWP::UserAgent;

my $ua = LWP::UserAgent->new;
p $ua;
```

Il examinera toutes les méthodes de l'objet et listera également les composants internes.

```
LWP::UserAgent  {
    Parents       LWP::MemberMixin
    public methods (45) : add_handler, agent, clone, conn_cache, cookie_jar, credentials,
default_header, default_headers, delete, env_proxy, from, get, get_basic_credentials,
get_my_handler, handlers, head, is_online, is_protocol_supported, local_address, max_redirect,
max_size, mirror, new, no_proxy, parse_head, post, prepare_request, progress,
protocols_allowed, protocols_forbidden, proxy, put, redirect_ok, remove_handler, request,
requests_redirectable, run_handlers, send_request, set_my_handler, show_progress,
simple_request, ssl_opts, timeout, use_alarm, use_eval
    private methods (4) : _agent, _need_proxy, _new_response, _process_colonic_headers
    internals: {
        def_headers             HTTP::Headers,
        handlers                {
            response_header   HTTP::Config
        },
        local_address           undef,
        max_redirect            7,
        max_size                undef,
        no_proxy                [],
        protocols_allowed       undef,
        protocols_forbidden     undef,
        proxy                   {},
        requests_redirectable   [
            [0] "GET",
            [1] "HEAD"
        ],
        show_progress           undef,
        ssl_opts                {
            verify_hostname   1
        },
        timeout                 180,
        use_eval                1
    }
}
```

Vous pouvez le configurer davantage, de sorte qu'il sérialise certains objets d'une certaine manière, ou d'inclure des objets jusqu'à une profondeur arbitraire. La configuration complète est disponible dans la documentation .

Malheureusement, Data :: Printer n'est pas livré avec Perl, vous devez donc l'installer à partir de CPAN ou via votre système de gestion de paquets.

Liste de tableaux de dumping

```
my @data_array = (123, 456, 789, 'poi', 'uyt', "rew", "qas");
print Dumper @data_array;
```

Utiliser Data :: Dumper permet d'accéder facilement aux valeurs de liste. Le `Dumper` renvoie les valeurs de liste sérialisées de manière à ressembler à du code Perl.

Sortie:

```
$VAR1 = 123;
$VAR2 = 456;
$VAR3 = 789;
$VAR4 = 'poi';
```

```
$VAR5 = 'uyt';
$VAR6 = 'rew';
$VAR7 = 'qas';
```

Comme suggéré par l'utilisateur @dgw Lors du vidage de tableaux ou de hachages, il est préférable d'utiliser une référence de tableau ou une référence de hachage, celles-ci seront mieux adaptées à l'entrée.

```
$ref_data = [23,45,67,'mnb','vcx'];
print Dumper $ref_data;
```

Sortie:

```
$VAR1 = [
          23,
          45,
          67,
          'mnb',
          'vcx'
        ];
```

Vous pouvez également référencer le tableau lors de l'impression.

```
my @data_array = (23,45,67,'mnb','vcx');
print Dumper \@data_array;
```

Sortie:

```
$VAR1 = [
          23,
          45,
          67,
          'mnb',
          'vcx'
        ];
```

Exposition des données

La fonction `show` est automatiquement exportée lorsque vous `use Data::Show;` est exécuté. Cette fonction prend une variable comme seul argument et génère:

1. Le nom de la variable
2. le contenu de cette variable (dans un format lisible)
3. la ligne du fichier qui `show` est exécutée à partir de
4. le fichier `show` est exécuté à partir de

En supposant que le code suivant provient du fichier `example.pl` :

```
use strict;
use warnings;
use Data::Show;
```

```
my @array = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

my %hash  = ( foo => 1, bar => { baz => 10, qux => 20 } );

my $href = \%hash;

show @array;
show %hash;
show $href;
```

`perl example.pl` donne la sortie suivante:

```
======(  @array  )======================[ 'example.pl', line 11 ]======

    [1 .. 10]


======(  %hash  )=======================[ 'example.pl', line 12 ]======

    { bar => { baz => 10, qux => 20 }, foo => 1 }


======(  $href  )=======================[ 'example.pl', line 13 ]======

    { bar => { baz => 10, qux => 20 }, foo => 1 }
```

Voir la documentation de `Data::Show` .

Lire Déboguer la sortie en ligne: https://riptutorial.com/fr/perl/topic/983/deboguer-la-sortie

Chapitre 14: Des listes

Examples

Tableau comme liste

Le tableau est l'un des types de variables de base de Perl. Il contient une liste, qui est une séquence ordonnée de zéro ou plusieurs scalaires. Le tableau est la variable contenant (et donnant accès aux) données de liste, comme cela est documenté dans perldata .

Vous pouvez affecter une liste à un tableau:

```
my @foo = ( 4, 5, 6 );
```

Vous pouvez utiliser un tableau partout où une liste est attendue:

```
join '-', ( 4, 5, 6 );
join '-', @foo;
```

Certains opérateurs ne fonctionnent qu'avec des tableaux car ils modifient la liste contenue dans un tableau:

```
shift @array;
unshift @array, ( 1, 2, 3 );
pop @array;
push @array, ( 7, 8, 9 );
```

Assigner une liste à un hachage

Les listes peuvent également être affectées à des variables de hachage. Lors de la création d'une liste qui sera assignée à une variable de hachage, il est recommandé d'utiliser la ***virgule grasse*** `=>` entre les clés et les valeurs pour afficher leur relation:

```
my %hash = ( foo => 42, bar => 43, baz => 44 );
```

Le `=>` n'est en réalité qu'une virgule spéciale qui cite automatiquement l'opérande à sa gauche. Donc, vous *pouvez* utiliser des virgules normales, mais la relation n'est pas aussi claire:

```
my %hash = ( 'foo', 42, 'bar', 43, 'baz', 44 );
```

Vous pouvez également utiliser des chaînes entre guillemets pour l'opérande gauche de la virgule grasse `=>` , ce qui est particulièrement utile pour les clés contenant des espaces.

```
my %hash = ( 'foo bar' => 42, 'baz qux' => 43 );
```

Pour plus de détails, voir Opérateur de virgule à `perldoc perlop` .

Les listes peuvent être transmises en sous-programmes

Pour transmettre la liste à un sous-programme, vous spécifiez le nom du sous-programme, puis vous lui fournissez la liste:

```
test_subroutine( 'item1', 'item2' );
test_subroutine  'item1', 'item2';     # same
```

En interne, Perl crée des *alias* pour ces arguments et les place dans le tableau `@_` disponible dans le sous-programme:

```
@_ =  ( 'item1', 'item2' ); # Done internally by perl
```

Vous accédez à des arguments de sous-routine comme ceci:

```
sub test_subroutine {
    print $_[0]; # item1
    print $_[1]; # item2
}
```

L'aliasing vous permet de modifier la valeur d'origine de l'argument passé à la sous-routine:

```
sub test_subroutine {
    $_[0] +=  2;
}

my $x =  7;
test_subroutine( $x );
print $x; # 9
```

Pour éviter les modifications involontaires des valeurs d'origine transmises à votre sous-routine, vous devez les copier:

```
sub test_subroutine {
    my( $copy_arg1, $copy_arg2 ) =  @_;
    $copy_arg1 += 2;
}

my $x =  7;
test_subroutine $x; # in this case $copy_arg2 will have `undef` value
print $x; # 7
```

Pour tester le nombre d'arguments passés dans le sous-programme, vérifiez la taille de `@_`

```
sub test_subroutine {
    print scalar @_, ' argument(s) passed into subroutine';
}
```

Si vous passez des arguments de tableau dans un sous-programme, ils seront tous *aplatis* :

```
my @x =  ( 1, 2, 3 );
my @y =  qw/ a b c /; # ( 'a', 'b', 'c' )
test_some_subroutine @x, 'hi', @y; # 7 argument(s) passed into subroutine
# @_ =  ( 1, 2, 3, 'hi', 'a', 'b', 'c' ) # Done internally for this call
```

Si votre `test_some_subroutine` contient l'instruction `$_[4] = 'd'` , pour l'appel ci-dessus, `$y[0]` aura la valeur `d` après:

```
print "@y"; # d b c
```

Liste de retour du sous-programme

Vous pouvez bien sûr renvoyer des listes de sous-marins:

```
sub foo {
    my @list1 =  ( 1, 2, 3 );
    my @list2 =  ( 4, 5 );

    return    ( @list1, @list2 );
}

my @list =  foo();
print @list;          # 12345
```

Mais ce n'est pas la façon recommandée de le faire à moins **que** vous sachiez ce que vous faites.

Bien que cela soit correct lorsque le résultat est dans un contexte **LIST** , dans le contexte **SCALAR** , les choses ne sont pas claires. Regardons la ligne suivante:

```
print scalar foo();  # 2
```

Pourquoi `2` ? Que se passe-t-il?

1. `foo()` évalué dans le contexte *SCALAR* , cette liste `( @list1, @list2 )` également évaluée dans le contexte *SCALAR*
2. Dans le contexte *SCALAR* , LIST renvoie son dernier élément. Ici c'est `@list2`
3. Toujours dans le contexte *SCALAR* , **array** `@list2` renvoie le nombre de ses éléments. Ici c'est `2` .

Dans la plupart des cas, la **bonne stratégie renverra des références aux structures de données** .
Donc, dans notre cas, nous devrions plutôt faire ce qui suit:

```
 return    ( \@list1, \@list2 );
```

Ensuite, l'appelant fait quelque chose comme ça pour recevoir les deux *tableaux* retournés:

```
 my ($list1, $list2) = foo(...);
```

Utiliser arrayref pour passer le tableau à sub

Le tableau RAF pour `@foo` est `\@foo` . C'est pratique si vous devez passer un tableau et d'autres choses à une sous-routine. Passer `@foo` c'est comme passer plusieurs scalaires. Mais passer `\@foo` est un scalaire unique. Dans la sous-routine:

```
xyz(\@foo, 123);
...
sub xyz {
    my ($arr, $etc) = @_;
    print $arr->[0]; # using the first item in $arr. It is like $foo[0]
```

Hash comme liste

Dans le contexte de la liste, le hachage est aplati.

```
my @bar =  ( %hash, %hash );
```

Le *tableau* `@bar` est initialisé par la liste de deux `%hash` hachages de `%hash`

- les deux `%hash` sont aplatis
- nouvelle liste est créée à partir d'éléments aplatis
- `@bar` tableau `@bar` est initialisé par cette liste

Il est garanti que les paires clé-valeur vont ensemble. Les clés sont toujours indexées, les valeurs - impaires. Il n'est pas garanti que les paires clé-valeur soient toujours aplaties dans le même ordre:

```
my %hash =  ( a => 1, b => 2 );
print %hash; # Maybe 'a1b2' or 'b2a1'
```

Lire Des listes en ligne: https://riptutorial.com/fr/perl/topic/4553/des-listes

Chapitre 15: Emballer et déballer

Examples

Conversion manuelle des structures C en syntaxe de pack

Si vous avez déjà affaire à des API C Binary à partir de Perl Code, via les fonctions `syscall` , `ioctl` ou `fcntl` , vous devez savoir comment construire de la mémoire à l'aide de C Compatible.

Par exemple, si vous étiez en `/usr/include/time.h` traiter avec une fonction qui attendait une `timespec` , vous devriez regarder dans `/usr/include/time.h` et trouver:

```
struct timespec
{
    __time_t tv_sec;            /* Seconds.  */
    __syscall_slong_t tv_nsec;  /* Nanoseconds.  */
};
```

Vous faites une danse avec `cpp` pour trouver ce que cela signifie vraiment:

```
cpp -E /usr/include/time.h -o /dev/stdout | grep __time_t
# typedef long int __time_t;
cpp -E /usr/include/time.h -o /dev/stdout | grep __syscall_slong_t
# typedef long int __syscall_slong_t
```

Donc c'est un int (signé)

```
echo 'void main(){ printf("%#lx\n", sizeof(__syscall_slong_t)); }' |
  gcc -x c -include stdio.h -include time.h - -o /tmp/a.out && /tmp/a.out
# 0x8
```

Et cela prend 8 octets. Donc, 64 bits signé int. Et je suis sur un processeur 64 bits. =)

Le `pack` Perldoc dit

```
            q  A signed quad (64-bit) value.
```

Donc, pour emballer un timespec:

```
sub packtime {
    my ( $config ) = @_;
    return pack 'qq', @{$config}{qw( tv_sec tv_nsec )};
}
```

Et pour décompresser une fois:

```
sub unpacktime {
   my ( $buf ) = @_;
```

```
    my $out = {};
    @{$out}{qw( tv_sec tv_nsec )} = unpack 'qq', $buf;
    return $out;
}
```

Maintenant, vous pouvez simplement utiliser ces fonctions à la place.

```
my $timespec = packtime({ tv_sec => 0, tv_nsec => 0 });
syscall(  ..., $timespec ); # some syscall that reads timespec

later ...
syscall( ..., $timespec ); # some syscall that writes timespec
print Dumper( unpacktime( $timespec ));
```

Construire un en-tête IPv4

Parfois, vous devez gérer des structures définies en termes de types de données C à partir de Perl. L'une de ces applications est la création de paquets réseau bruts, au cas où vous souhaiteriez faire quelque chose de plus sophistiqué que ce que l'API de socket classique peut offrir. Ceci est juste ce que `pack()` (et `unpack()` bien sûr) est là pour.

La partie obligatoire d'un en-tête IP est longue de 20 octets (AKA "octets"). Comme vous pouvez le voir derrière ce lien, les adresses IP source et destination constituent les deux dernières valeurs 32 bits de l'en-tête. Parmi les autres champs, il y en a avec 16 bits, certains avec 8 bits et quelques petits morceaux entre 2 et 13 bits.

En supposant que nous ayons les variables suivantes à remplir dans notre en-tête:

```
my ($dscp, $ecn, $length,
    $id, $flags, $frag_off,
    $ttl, $proto,
    $src_ip,
    $dst_ip);
```

Notez que trois champs de l'en-tête sont manquants:

- La version est toujours 4 (c'est IPv4 après tout)
- Le DIH est de 5 dans notre exemple car nous n'avons pas de champ d' *options* ; la longueur est spécifiée en unités de 4 octets, donc 20 octets donnent une longueur de 5.
- La somme de contrôle peut être laissée à 0. En fait, il faudrait le calculer, mais le code pour cela ne nous concerne pas ici.

Nous pourrions essayer d'utiliser des opérations sur les bits pour construire par exemple les 32 premiers bits:

```
my $hdr = 4 << 28 | 5 << 24 | $dscp << 18 | $ecn << 16 | $length;
```

Cette approche ne fonctionne que jusqu'à la taille d'un entier mais, ce qui est généralement 64 bits mais peut être aussi bas que 32. Pire encore, cela dépend de la CPU de la boutisme il fonctionnera sur certains processeurs et ne parviennent pas à d' autres. Essayons `pack()` :

```
my $hdr = pack('H2B8n', '45', sprintf("%06b%02b", $dscp, $ecn), $length);
```

Le modèle spécifie d'abord `H2` , une *chaîne hexadécimale à deux caractères, nybble haut en premier* . L'argument correspondant à pack est "45" - version 4, longueur 5. Le modèle suivant est `B8` , une *chaîne de bits de 8 bits, ordre des bits décroissant à l'intérieur de chaque octet* . Nous devons utiliser des chaînes de bits pour contrôler la mise en page jusqu'à des morceaux plus petits qu'un nybble (4 bits), de sorte que le `sprintf()` soit utilisé pour construire une chaîne de 6 bits à partir de `$dscp` et 2 de `$ecn` . Le dernier est `n` , une *valeur de 16 bits non signée dans Network Byte Order* , c'est-à-dire toujours big-endian quel que soit le format entier natif de votre CPU, et il est rempli à partir de `$length` .

Ce sont les 32 premiers bits de l'en-tête. Le reste peut être construit de la même manière:

Modèle	Argument	Remarques
`n`	`$id`	
`B16`	`sprintf("%03b%013b", $flags, $frag_off)`	Identique à DSCP / ECN
`C2`	`$ttl, $proto`	Deux octets non signés consécutifs
`n`	`0` / `$checksum`	`x` pourrait être utilisé pour insérer un octet nul mais `n` nous permet de spécifier un argument si nous choisissons de calculer une somme de contrôle
`N2`	`$src_ip, $dst_ip`	Utilisez `a4a4` pour `a4a4` le résultat de deux appels `gethostbyname()` tels qu'ils sont dans l'ordre des octets de réseau!

L'appel complet pour empaqueter un en-tête IPv4 serait:

```
my $hdr = pack('H2B8n2B16C2nN2',
    '45', sprintf("%06b%02b", $dscp, $ecn), $length,
    $id, sprintf("%03b%013b", $flags, $frag_off),
    $ttl, $proto, 0,
    $src_ip, $dst_ip
);
```

Lire Emballer et déballer en ligne: https://riptutorial.com/fr/perl/topic/1983/emballer-et-deballer

Chapitre 16: Expressions régulières

Examples

Chaînes correspondantes

L'opérateur `=~` tente de faire correspondre une expression régulière (définie par `/`) à une chaîne:

```
my $str = "hello world";
print "Hi, yourself!\n" if $str =~ /^hello/;
```

`/^hello/` est l'expression régulière réelle. Le `^` est un caractère spécial qui dit à l'expression régulière de commencer avec le début de la chaîne et de ne pas correspondre au milieu quelque part. Ensuite, les regex essaient de trouver les lettres suivantes dans l'ordre `h` , `e` , `l` , `l` et `o` .

Les expressions régulières tentent de faire correspondre la variable par défaut (`$_`) si elle est vide:

```
$_ = "hello world";

print "Ahoy!\n" if /^hello/;
```

Vous pouvez également utiliser différents délimiteurs si vous faites précéder l'expression régulière avec l'opérateur `m` :

```
m~^hello~;
m{^hello};
m|^hello|;
```

Ceci est utile pour faire correspondre les chaînes qui incluent le caractère `/` :

```
print "user directory" if m|^/usr|;
```

Utilisation de \ Q et \ E dans la correspondance de modèle

Ce qui est entre \ Q et \ E est traité comme des caractères normaux

```
#!/usr/bin/perl

my $str = "hello.it's.me";

my @test = (
  "hello.it's.me",
    "hello/it's!me",
    );
```

```
sub ismatched($) { $_[0] ? "MATCHED!" : "DID NOT MATCH!" }

my @match = (
      [ general_match=> sub { ismatched /$str/ } ],
      [ qe_match    => sub { ismatched /\Q$str\E/ } ],
      );

for (@test) {
    print "\String = '$_':\n";

foreach my $method (@match) {
    my($name,$match) = @$method;
    print "  - $name: ", $match->(), "\n";
}
```

}

Sortie

```
String = 'hello.it's.me':
  - general_match: MATCHED!
  - qe_match: MATCHED!
String = 'hello/it's!me':
  - general_match: MATCHED!
  - qe_match: DID NOT MATCH!
```

Analyser une chaîne avec une expression régulière

Généralement, ce n'est pas une bonne idée d' utiliser une expression régulière pour analyser une structure complexe . Mais cela peut être fait. Par exemple, vous pouvez charger des données dans la table de ruche et les champs sont séparés par des virgules, mais les types complexes comme les tableaux sont séparés par un "|". Les fichiers contiennent des enregistrements dont tous les champs sont séparés par des virgules et des types complexes. Dans ce cas, ce bit de Perl jetable pourrait suffire:

```
echo "1,2,[3,4,5],5,6,[7,8],[1,2,34],5" | \
    perl -ne \
        'while( /\[[^,\]]+\,.*\]/ ){
            if( /\[([^\]\|]+)\]/){
                $text = $1;
                $text_to_replace = $text;
                $text =~ s/\,/\|/g;
                s/$text_to_replace/$text/;
            }
        } print'
```

Vous voudrez vérifier la sortie:

1,2, [3 | 4 | 5], 5,6, [7 | 8], [1 | 2 | 34], 5

Remplacer une chaîne à l'aide d'expressions régulières

```
s/foo/bar/;         # replace "foo" with "bar" in $_
my $foo = "foo";
$foo =~ s/foo/bar/; # do the above on a different variable using the binding operator =~
s~ foo ~ bar ~;     # using ~ as a delimiter
$foo = s/foo/bar/r; # non-destructive r flag: returns the replacement string without modifying
the variable it's bound to
s/foo/bar/g;        # replace all instances
```

Lire Expressions régulières en ligne: https://riptutorial.com/fr/perl/topic/3108/expressions-regulieres

Chapitre 17: Fichier I / O (lecture et écriture de fichiers)

Paramètres

Mode	Explication
`>`	**Ecrire (trunc)** . Ecrasera les fichiers existants. Crée un nouveau fichier si aucun fichier n'a été trouvé
`>>`	**Ecrire (ajouter)** . N'écrasera pas les fichiers mais ajoutera du nouveau contenu à la fin. Crée également un fichier s'il est utilisé pour ouvrir un fichier non existant.
`<`	**Lire** . Ouvre le fichier en mode lecture seule.
`+<`	**Lecture / écriture** . Ne va pas créer ou tronquer le fichier.
`+>`	**Lecture / écriture (trunc)** . Va créer et tronquer le fichier.
`+>>`	**Lecture / écriture (append)** . Va créer mais ne pas tronquer le fichier.

Remarques

`chomp` est souvent utilisé lors de la lecture d'un fichier. Par défaut, il coupe le caractère de nouvelle ligne, bien que pour sa fonctionnalité complète, reportez-vous aux perldocs .

Méfiez-vous de la différence entre les caractères et les octets: tous les encodages, en particulier UTF-8, n'utilisent que des caractères d'un octet. Bien que PerlIO gère parfaitement ce problème, il existe un écueil potentiel:

- `read` utilise des *caractères* pour ses paramètres de *longueur* et de *décalage*
- `seek` et `tell` *toujours* utiliser des *octets* pour le positionnement

N'utilisez donc pas d'arithmétique basée sur ces valeurs mixtes. Utilisez plutôt, par exemple, `Encode::encode('utf8',$value_by_read)` pour obtenir les octets (octets) d'un résultat de `read` , dont vous pouvez alors utiliser le compte avec `tell` et `seek` .

Examples

Lecture d'un fichier

```
my $filename = '/path/to/file';

open my $fh, '<', $filename or die "Failed to open file: $filename";
```

```
# You can then either read the file one line at a time...
while(chomp(my $line = <$fh>)) {
    print $line . "\n";
}

# ...or read whole file into an array in one go
chomp(my @fileArray = <$fh>);
```

Si vous savez que votre fichier d'entrée est UTF-8, vous pouvez spécifier l'encodage:

```
open my $fh, '<:encoding(utf8)', $filename or die "Failed to open file: $filename";
```

Une fois la lecture du fichier terminée, le descripteur de fichier doit être fermé:

```
close $fh or warn "close failed: $!";
```

Voir aussi: Lecture d'un fichier dans une variable

Un autre moyen **plus rapide** de lire un fichier consiste à utiliser File :: Slurper Module. Ceci est utile si vous travaillez avec de nombreux fichiers.

```
use File::Slurper;
my $file = read_text("path/to/file"); # utf8 without CRLF transforms by default
print $file; #Contains the file body
```

Voir aussi: [Lire un fichier avec slurp]

Ecrire dans un fichier

Ce code ouvre un fichier pour l'écriture. Renvoie une erreur si le fichier n'a pas pu être ouvert. Ferme également le fichier à la fin.

```
#!/usr/bin/perl
use strict;
use warnings;
use open qw( :encoding(UTF-8) :std ); # Make UTF-8 default encoding

# Open "output.txt" for writing (">") and from now on, refer to it as the variable $fh.
open(my $fh, ">", "output.txt")
# In case the action failed, print error message and quit.
or die "Can't open > output.txt: $!";
```

Nous avons maintenant un fichier ouvert prêt à être écrit auquel nous accédons via `$fh` (cette variable s'appelle un *descripteur de* fichier). Ensuite, nous pouvons diriger la sortie vers ce fichier en utilisant l'opérateur d' `print` :

```
# Print "Hello" to $fh ("output.txt").
print $fh "Hello";
# Don't forget to close the file once we're done!
close $fh or warn "Close failed: $!";
```

L'opérateur `open` a une variable scalaire (`$fh` dans ce cas) comme premier paramètre. Comme il est défini dans l'opérateur `open` , il est traité comme un *descripteur de fichier* . Le deuxième paramètre `">"` (supérieur à) définit que le fichier est ouvert pour écriture. Le dernier paramètre est le chemin du fichier dans lequel écrire les données.

Pour écrire les données dans le fichier, l'opérateur d' `print` est utilisé avec le *descripteur de* fichier. Notez que dans l'opérateur d' `print` , il n'y a pas de virgule entre le *descripteur de fichier* et l'instruction elle-même, juste un espace.

Ouvrir un fichier FileHandle pour la lecture

Ouverture de fichiers texte ASCII génériques

5.6.0

```
open my $filehandle, '<', $name_of_file or die "Can't open $name_of_file, $!";
```

Ceci est l'idiome de base pour "défaut" File IO et fait de `$filehandle` un flux d' `bytes` entrée lisible, filtré par un décodeur spécifique au système par défaut, qui peut être défini localement avec le pragma `open`

Perl lui-même ne gère pas les erreurs lors de l'ouverture des fichiers. Vous devez donc les gérer vous-même en vérifiant la condition d' `open` de `open` . `$!` est rempli avec le message d'erreur à l'origine de l'échec.

Sous Windows, le décodeur par défaut est un filtre "CRLF", qui mappe toutes les séquences "\ r \ n" de l'entrée à "\ n"

Ouverture de fichiers binaires

5.8.0

```
open my $filehandle, '<:raw', 'path/to/file' or die "Can't open $name_of_file, $!";
```

Cela spécifie que Perl *ne* doit *pas* effectuer de traduction `CRLF` sous Windows.

Ouverture de fichiers texte UTF8

5.8.0

```
open my $filehandle, '<:raw:encoding(utf-8)', 'path/to/file'
   or die "Can't open $name_of_file, $!";
```

Cela spécifie que Perl doit à la fois éviter la traduction `CRLF` , puis décoder les octets résultants en chaînes de *caractères* (implémentées en interne sous la forme de tableaux d'entiers pouvant dépasser 255), au lieu de chaînes d' *octets*

Lecture et écriture dans un fichier

Avant de lire et d'écrire des fichiers texte, vous devez savoir quel encodage utiliser. Consultez la documentation Perl Unicode pour plus de détails sur le codage . Nous montrons ici le réglage de UTF-8 comme codage et décodage par défaut pour la fonction `open` . Cela se fait en utilisant le pragma `open` haut de votre code (juste après l' `use strict;` et `use warnings;` serait approprié):

```
use strict;
use warnings;
use open qw( :encoding(UTF-8) :std );
```

La fonction `open` crée un descripteur de fichier utilisé pour lire et / ou écrire dans un fichier. La fonction `open` a la signature

`open(FILEHANDLE, MODE, FILEPATH)` et renvoie une valeur fausse si l'opération échoue. La description de l'erreur est ensuite stockée dans `$!` .

En train de lire

```
#!/usr/bin/perl
use strict;
use warnings;
use open qw( :encoding(UTF-8) :std ); # Make UTF-8 default encoding

my $file_path = "/path/to/file";
open(my $file_handle, '<', $file_path) or die "Could not open file! $!";

while(my $row = <$file_handle>) {
    print chomp($row), "\n";
}

close $file_handle
      or warn "Close failed!";
```

L'écriture

```
#!/usr/bin/perl
use strict;
use warnings;
use open qw( :encoding(UTF-8) :std ); # Make UTF-8 default encoding

my $file_path = "/path/to/file";
open(my $file_handle, '>', $file_path) or die "Could not open file! $!";

print $file_handle "Writing to a file";

close $file_handle
      or warn "Close failed!";
```

Lecture de morceaux

L'ouverture et la lecture de gros fichiers peuvent prendre du temps et des ressources. Si seulement une petite partie du contenu est requise, il peut être judicieux de lire le contenu en

morceaux à l'aide de la fonction de `read` qui a la signature.

```
read(FILEHANDLE, SCALAR, LENGTH, OFFSET)
```

`FILEHANDLE` doit être un handle de fichier ouvert, `SCALAR` conservera les données lues après l'opération. `LENGTH` spécifie le nombre de caractères à lire à partir du `OFFSET` . La fonction renvoie le nombre de caractères lus, `0` si la fin du fichier a été atteinte et `undef` en cas d'erreur.

```
read($file_handle, $data, 16, 0);
```

Lit 16 caractères depuis le début du fichier dans `$data` .

"utiliser autodie" et vous n'avez pas besoin de vérifier les échecs d'ouverture / fermeture de fichier

`autodie` vous permet de travailler avec des fichiers sans avoir à vérifier explicitement les échecs d'ouverture / fermeture.

Depuis Perl 5.10.1, le pragma `autodie` est disponible dans le noyau Perl. Lorsqu'il est utilisé, Perl vérifie automatiquement les erreurs lors de l'ouverture et de la fermeture des fichiers.

Voici un exemple dans lequel toutes les lignes d'un fichier sont lues puis écrites à la fin d'un fichier journal.

```
use 5.010;    # 5.010 and later enable "say", which prints arguments, then a newline
use strict;   # require declaring variables (avoid silent errors due to typos)
use warnings; # enable helpful syntax-related warnings
use open qw( :encoding(UTF-8) :std ); # Make UTF-8 default encoding
use autodie;  # Automatically handle errors in opening and closing files

open(my $fh_in, '<', "input.txt"); # check for failure is automatic

# open a file for appending (i.e. using ">>")
open( my $fh_log, '>>', "output.log"); # check for failure is automatic

while (my $line = readline $fh_in) # also works: while (my $line = <$fh_in>)
{
     # remove newline
     chomp $line;

     # write to log file
     say $fh_log $line or die "failed to print '$line'"; # autodie doesn't check print
}

# Close the file handles (check for failure is automatic)
close $fh_in;
close $fh_log;
```

En passant, vous devriez toujours vérifier techniquement `print` instructions `print` . Beaucoup de gens ne le font pas, mais `perl` (l'interpréteur Perl) ne le fait pas automatiquement, pas plus `autodie` .

Rembobiner un descripteur de fichier

Parfois, il est nécessaire de revenir en arrière après avoir lu.

```
# identify current position in file, in case the first line isn't a comment
my $current_pos = tell;

while (my $line = readline $fh)
{
    if ($line =~ /$START_OF_COMMENT_LINE/)
    {
        push @names, get_name_from_comment($line);
    }
    else {
        last; # break out of the while loop
    }
    $current_pos = tell; # keep track of current position, in case we need to rewind the next
line read
}

# Step back a line so that it can be processed later as the first data line
seek $fh, $current_pos, 0;
```

Lecture et écriture de fichiers compressés gzip

Ecrire un fichier compressé

Pour écrire un fichier compressé, `use` le module `IO::Compress::Gzip` et créez un descripteur de fichier en créant une nouvelle instance de `IO::Compress::Gzip` pour le fichier de sortie souhaité:

```
use strict;
use warnings;
use open qw( :encoding(UTF-8) :std ); # Make UTF-8 default encoding

use IO::Compress::Gzip;

my $fh_out = IO::Compress::Gzip->new("hello.txt.gz");

print $fh_out "Hello World!\n";

close $fh_out;

use IO::Compress::Gzip;
```

Lecture d'un fichier compressé

Pour lire un fichier compressé, `use` le module `IO::Uncompress::Gunzip` , puis créez un descripteur de fichier en créant une nouvelle instance de `IO::Uncompress::Gunzip` pour le fichier d'entrée:

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

```
use warnings;
use open qw( :encoding(UTF-8) :std ); # Make UTF-8 default encoding

use IO::Uncompress::Gunzip;

my $fh_in = IO::Uncompress::Gunzip->new("hello.txt.gz");

my $line = readline $fh_in;

print $line;
```

Définition du codage par défaut pour IO

```
# encode/decode UTF-8 for files and standard input/output
use open qw( :encoding(UTF-8) :std );
```

Ce `pragma` change le mode par défaut de lecture et d'écriture du texte (fichiers, entrée standard, sortie standard et erreur standard) en UTF-8, ce qui correspond généralement à ce que vous souhaitez lors de l'écriture de nouvelles applications.

ASCII est un sous-ensemble de UTF-8, ce qui ne devrait pas poser de problème avec les fichiers ASCII hérités et vous aidera à vous protéger contre la corruption accidentelle de fichiers pouvant survenir lors du traitement de fichiers UTF-8 en ASCII.

Cependant, il est important que vous sachiez quel est l'encodage de vos fichiers et que vous les gériez en conséquence. (Raisons pour lesquelles nous ne devrions pas ignorer Unicode.) Pour un traitement plus approfondi d'Unicode, veuillez consulter la rubrique Perl Unicode .

Lire Fichier I / O (lecture et écriture de fichiers) en ligne:
https://riptutorial.com/fr/perl/topic/1604/fichier-i---o--lecture-et-ecriture-de-fichiers-

Chapitre 18: Gestion des exceptions

Examples

eval et mourir

C'est la manière intégrée de gérer les "exceptions" sans avoir recours à des bibliothèques tierces comme Try :: Tiny .

```
my $ret;

eval {
  $ret = some_function_that_might_die();
  1;
} or do {
  my $eval_error = $@ || "Zombie error!";
  handle_error($eval_error);
};

# use $ret
```

Nous "abusons" du fait que `die` a une valeur de retour fausse, et la valeur de retour du bloc de code global est la valeur de la dernière expression dans le bloc de code:

- si `$ret` est assigné avec succès, alors le `1;` expression est la dernière chose qui se produit dans le bloc de code `eval` . Le bloc de code `eval` a donc une valeur vraie, donc le bloc `or do` ne s'exécute pas.
- Si `some_function_that_might_die() die` , la dernière chose qui se passe dans le bloc de code `eval` est le `die` . Le bloc de code `eval` a donc une valeur fausse et le bloc `or do` est exécuté.
- La première chose que vous *devez* faire dans le `or do` bloc est lu `$@` pour . Cette variable globale contiendra tout argument passé pour `die` . Le `|| "Zombie Error"` guard est populaire, mais inutile dans le cas général.

Ceci est important à comprendre car certains appels au dé ne font pas échouer tous les codes, mais la même structure peut être utilisée indépendamment. Considérons une fonction de base de données qui renvoie:

- le nombre de lignes affectées par le succès
- `'0 but true'` si la requête est réussie mais qu'aucune ligne n'a été affectée
- `0` si la requête a échoué.

Dans ce cas, vous pouvez toujours utiliser le même idiome, mais vous *devez* ignorer le dernier `1;` , et cette fonction *doit* être la dernière chose dans l'éval. Quelque chose comme ça:

```
eval {
  my $value = My::Database::retrieve($my_thing); # dies on fail
  $value->set_status("Completed");
  $value->set_completed_timestamp(time());
```

```
  $value->update(); # returns false value on fail
} or do { # handles both the die and the 0 return value
  my $eval_error = $@ || "Zombie error!";
  handle_error($eval_error);
};
```

Lire Gestion des exceptions en ligne: https://riptutorial.com/fr/perl/topic/1894/gestion-des-exceptions

Chapitre 19: Installation de Perl

Introduction

Je vais commencer par le processus dans Ubuntu, puis dans OS X et enfin dans Windows. Je ne l'ai pas testé sur toutes les versions de perl, mais cela devrait être un processus similaire.

Utilisez Perlbrew si vous souhaitez basculer facilement entre les différentes versions de Perl.

Je tiens à préciser que ce tutoriel concerne Perl dans sa version open-source. Il existe d'autres versions comme `activeperl` dont les avantages et les inconvénients ne font pas partie de ce tutoriel.

Examples

Linux

Il y a plus d'une façon de le faire:

- En utilisant le gestionnaire de paquets:

```
sudo apt install perl
```

- Installation à partir de la source:

```
wget http://www.cpan.org/src/5.0/perl-version.tar.gz
tar -xzf perl-version.tar.gz
cd perl-version
./Configure -de
make
make test
make install
```

- Installer dans votre répertoire $ home (pas sudo nécessaire) avec Perlbrew :

```
wget -O - https://install.perlbrew.pl | bash
```

  Voir aussi Perlbrew

OS X

Il y a plusieurs options:

- Perlbrew :

```
# You need to install Command Line Tools for Xcode
curl -L https://install.perlbrew.pl | bash
```

- Perlbrew avec support de thread:

```
# You need to install Command Line Tools for Xcode
curl -L https://install.perlbrew.pl | bash
```

Après l'installation de perlbrew, si vous souhaitez installer Perl avec un support de thread, exécutez simplement:

```
perlbrew install -v perl-5.26.0 -Dusethreads
```

- De source:

```
tar -xzf perl-version.tar.gz
cd perl-version
./Configure -de
make
make test
make install
```

les fenêtres

- Comme nous l'avons déjà dit, nous allons avec la version open source. Pour Windows, vous pouvez choisir la `strawberry` ou le `DWIM` . Ici, nous couvrons la version `strawberry` , puisque `DWIM` est basé dessus. La méthode la plus simple consiste à installer depuis l' exécutable officiel .

Voir aussi berrybrew - le perlbrew pour Windows Strawberry Perl

Lire Installation de Perl en ligne: https://riptutorial.com/fr/perl/topic/9317/installation-de-perl

Chapitre 20: Installer les modules Perl via CPAN

Examples

Exécutez le CPAN Perl dans votre terminal (Mac et Linux) ou dans l'invite de commande (Windows)

Ligne de commande

Vous pouvez utiliser `cpan` pour installer des modules directement à partir de la ligne de commande:

```
cpan install DBI
```

Cela pourrait être suivi par de nombreuses pages de sortie décrivant exactement ce qu'il fait pour installer le module. Selon les modules installés, il est possible qu'il se mette en pause et vous pose des questions.

Shell interactif

Vous pouvez également entrer un "shell" ainsi:

```
perl -MCPAN -e "shell"
```

Il produira une sortie comme ci-dessous:

```
Terminal does not support AddHistory.

cpan shell -- CPAN exploration and modules installation (v2.00)
Enter 'h' for help.

cpan[1]>
```

Ensuite, vous pouvez installer les modules que vous souhaitez par la simple commande `install <module>` .

Exemple: `cpan[1]> install DBI`

Après l'installation, tapez `exit` pour quitter.

Installer les modules manuellement

Si vous ne disposez pas des autorisations nécessaires pour installer les modules Perl, vous pouvez toujours les installer manuellement, en indiquant un chemin d'accès personnalisé sur lequel vous disposez des droits d'écriture.

Poing, télécharger et décompresser l'archive du module:

```
wget module.tar.gz
tar -xzf module.tar.gz
cd module
```

Ensuite, si la distribution du module contient un fichier `Makefile.PL` , exécutez:

```
perl Makefile.PL INSTALL_BASE=$HOME/perl
make
make test
make install
```

ou si vous avez `Build.PL` fichier `Build.PL` au lieu d'un `Makefile.PL` :

```
perl Build.PL --install_base $HOME/perl
perl Build
perl Build test
perl Build install
```

Vous devez également inclure le chemin du module dans la variable d'environnement `PERL5LIB` afin de l'utiliser dans votre code:

```
export PERL5LIB=$HOME/perl
```

cpanminus, le remplacement léger sans configuration de cpan

Usage

Pour installer un module (en supposant que `cpanm` soit déjà installé):

```
cpanm Data::Section
```

`cpanm` ("cpanminus") s'efforce d'être moins prolixe que `cpan` mais capture toujours toutes les informations d'installation dans un fichier journal au cas où il serait nécessaire. Il traite également de nombreuses "questions interactives" pour vous, contrairement à `cpan` .

`cpanm` est également populaire pour installer des dépendances d'un projet à partir, par exemple, GitHub. L'utilisation typique est de commencer par `cd` dans la racine du projet, puis d'exécuter

```
cpanm --installdeps .
```

Avec `--installdeps` il va:

1. Analyser et installer *configure_requires* dépend des deux

    - META.json
    - META.yml (si META.json est manquant)
2. Construisez le projet (équivalent à `perl Build.PL`), générant des fichiers MYMETA
3. L'analyse et l'installation *nécessite des* dépendances
    - MYMETA.json
    - MYMETA.yml (si MYMETA.json est manquant)

Pour spécifier le fichier "some.cpanfile", contenant les dépendances, exécutez:

```
cpanm --installdeps --cpanfile some.cpanfile .
```

Installation de `cpanm`

Il y a plusieurs façons de l'installer . Voici l'installation via `cpan` :

```
cpan App::cpanminus
```

Configuration de `cpanm`

Il n'y a **aucun** fichier de configuration pour `cpanm` . Il s'appuie plutôt sur les variables d'environnement suivantes pour sa configuration:

- `PERL_CPANM_OPT` (`PERL_CPANM_OPT` générales de la ligne de commande cpanm)
    - `export PERL_CPANM_OPT="--prompt"` # dans .bashrc, pour activer les invites, par exemple
    - `setenv PERL_CPANM_OPT "--prompt"` # dans .tcshrc
- `PERL_MM_OPT` (options de ligne de commande ExtUtils :: MakeMaker, affectent la cible d'installation du module)
- `PERL_MB_OPT` (Options de ligne de commande de Module :: Build, affecte la cible d'installation du module)

Lire Installer les modules Perl via CPAN en ligne: https://riptutorial.com/fr/perl/topic/3542/installer-les-modules-perl-via-cpan

Chapitre 21: Instructions de contrôle

Examples

Conditionnels

Perl prend en charge de nombreux types d'instructions conditionnelles (instructions basées sur des résultats booléens). Les instructions conditionnelles les plus courantes sont if-else, sauf les instructions ternaires. `given` instructions `given` sont introduites comme une construction de type C à partir de langages dérivés de C et sont disponibles dans les versions Perl 5.10 et supérieures.

Déclarations If-Else

La structure de base d'une instruction if est la suivante:

```
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ...
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK
```

Pour les instructions if simples, le if peut précéder ou succéder au code à exécuter.

```
$number = 7;
if ($number > 4) { print "$number is greater than four!"; }

# Can also be written this way
print "$number is greater than four!" if $number > 4;
```

Boucles

Perl prend en charge de nombreux types de constructions de boucle: for / foreach, while / do-while, et jusqu'à.

```
@numbers = 1..42;
for (my $i=0; $i <= $#numbers; $i++) {
    print "$numbers[$i]\n";
}

#Can also be written as
foreach my $num (@numbers) {
    print "$num\n";
}
```

La boucle while évalue le conditionnel *avant d'* exécuter le bloc associé. Donc, parfois, le bloc n'est jamais exécuté. Par exemple, le code suivant ne serait jamais exécuté si le descripteur `$fh` fichier `$fh` était le descripteur de fichier d'un fichier vide ou s'il était déjà épuisé avant le conditionnel.

```
while (my $line = readline $fh) {
    say $line;
}
```

En revanche, les boucles `do` / `while` et `do` / `until do` évaluer la condition *après* chaque exécution du bloc. Ainsi, une boucle `do` / `while` ou `do` / `until` est toujours exécutée au moins une fois.

```
my $greeting_count = 0;
do {
    say "Hello";
    $greeting_count++;
} until ( $greeting_count > 1)

# Hello
# Hello
```

Lire Instructions de contrôle en ligne: https://riptutorial.com/fr/perl/topic/4896/instructions-de-controle

Chapitre 22: Interaction simple avec la base de données via le module DBI

Paramètres

Colonne	Colonne
`$driver`	Pilote pour DB, "Pg" pour Postgresql et "mysql" pour MySQL
`$database`	le nom de votre base de données
`$userid`	votre identifiant de base de données
`$password`	votre mot de passe de base de données
`$query`	mettez votre requête ici, ex: "select * from $ your_table"

Examples

Module DBI

Vous devez vous assurer que le module DBI a été installé sur votre PC, puis suivez les étapes ci-dessous:

1. utiliser le module DBI dans votre script perl

```
use DBI;
```

2. Déclarez certains paramètres primaires

```
my $driver = "MyDriver";

my $database = "DB_name";

my $dsn = "DBI:$driver:dbname=$database";

my $userid = "your_user_ID";

my $password = "your_password";

my $tablename = "your_table";
```

3. Connectez-vous à votre base de données

```
my $dbh = DBI->connect($dsn, $userid, $password);
```

4. Préparez votre requête

```
my $query = $dbh->prepare("Your DB query");
```

Ex:

```
$my_query = qq/SELECT * FROM table WHERE column1 = 2/;

my $query = $dbh->prepare($my_query);
```

Nous pouvons également utiliser la variable dans la requête, comme ci-dessous:

```
my $table_name = "table";

my $filter_value = 2;

$my_query = qq/SELECT * FROM $table_name WHERE column1 = $filter_value/;
```

5. Exécutez votre requête

```
$query->execute();
```

* Remarque: Pour éviter une attaque par injection, vous devez utiliser des espaces réservés `?` au lieu de mettre votre variable dans la requête.

Ex: vous voulez afficher toutes les données de 'table' où column1 = $ value1 et column2 = $ value2:

```
my $query = $dbh->prepare("SELECT * FROM table WHERE column1 = ? AND column2 = ?;");

$query->execute($value1, $value2);
```

6. Fletch vos données

`my @row = $query->fetchrow_array();` stocker des données sous forme de tableau

ou

`my $ref = $sth->fetchrow_hashref();` stocker des données en tant que référence de hachage

7. Terminer et déconnecter la base de données

```
$sth->finish;

$dbh->disconnect();
```

Lire Interaction simple avec la base de données via le module DBI en ligne: https://riptutorial.com/fr/perl/topic/5917/interaction-simple-avec-la-base-de-donnees-via-le-module-dbi

Chapitre 23: Interpolation en Perl

Examples

Interpolation de base

L'interpolation signifie que l'interpréteur Perl remplacera les valeurs des variables pour leur nom et certains symboles (qui sont impossibles ou difficiles à saisir directement) pour les séquences de caractères spéciales (il est également appelé échapper). La distinction la plus importante est entre les guillemets simples et doubles: les guillemets doubles interpolent la chaîne incluse, mais pas les guillemets simples.

```
my $name = 'Paul';
my $age = 64;
print "My name is $name.\nI am $age.\n"; # My name is Paul.
                                         # I am 64.
```

Mais:

```
print 'My name is $name.\nI am $age.\n'; # My name is $name.\nI am $age.\n
```

Vous pouvez utiliser `q{}` (avec n'importe quel délimiteur) au lieu de guillemets simples et `qq{}` au lieu de guillemets doubles. Par exemple, `q{I'm 64}` permet d'utiliser une apostrophe dans une chaîne non interpolée (sinon, elle mettrait fin à la chaîne).

Déclarations:

```
print qq{$name said: "I'm $age".}; # Paul said: "I'm 64".
print "$name said: \"I'm $age\"."  # Paul said: "I'm 64".
```

Faites la même chose, mais dans le premier cas, vous n'avez pas besoin d'échapper aux guillemets dans la chaîne.

Si votre nom de variable est en conflit avec le texte environnant, vous pouvez utiliser la syntaxe `${var}` pour désambiguïser:

```
my $decade = 80;
print "I like ${decade}s music!"  # I like 80s music!
```

Ce qui est interpolé

Perl interpole les noms de variables:

```
my $name = 'Paul';
print "Hello, $name!\n"; # Hello, Paul!

my @char = ('a', 'b', 'c');
```

```
print "$char[1]\n"; # b

my %map = (a => 125, b => 1080, c => 11);
print "$map{a}\n"; # 125
```

Les tableaux peuvent être interpolés dans leur ensemble, leurs éléments sont séparés par des espaces:

```
my @char = ('a', 'b', 'c');
print "My chars are @char\n"; # My chars are a b c
```

Perl n'interpole *pas les* hachages dans leur ensemble:

```
my %map = (a => 125, b => 1080, c => 11);
print "My map is %map\n"; # My map is %map
```

et appels de fonction (y compris les constantes):

```
use constant {
    PI => '3.1415926'
};
print "I like PI\n";         # I like PI
print "I like " . PI . "\n"; # I like 3.1415926
```

Perl interpole les *séquences d'échappement* commençant par `\` :

```
\t                  horizontal tab
\n                  newline
\r                  return
\f                  form feed
\b                  backspace
\a                  alarm (bell)
\e                  escape
```

L'interpolation de `\n` dépend du système sur lequel le programme fonctionne: il produira un ou plusieurs caractères de nouvelle ligne conformément aux conventions système actuelles.

Perl n'interpole *pas* `\v` , ce qui signifie un onglet vertical en C et d'autres langages.

Le caractère peut être traité en utilisant leurs codes:

```
\x{1d11e}     by hexadecimal code
\o{350436}    by octal code
\N{U+1d11e}   by Unicode code point
```

ou noms Unicode:

```
\N{MUSICAL SYMBOL G CLEF}
```

Caractère avec des codes de `0x00` à `0xFF` dans le codage *natif* peut être adressé sous une forme plus courte:

```
\x0a     hexadecimal
\012     octal
```

Le caractère de contrôle peut être adressé à l'aide de séquences d'échappement spéciales:

```
\c@      chr(0)
\ca      chr(1)
\cb      chr(2)
...
\cz      chr(26)
\c[      chr(27)
\c\      chr(28) # Cannot be used at the end of a string
                 # since backslash will interpolate the terminating quote
\c]      chr(29)
\c^      chr(30)
\c_      chr(31)
\c?      chr(127)
```

Les lettres majuscules ont la même signification: `"\cA" == "\ca"` .

L'interprétation de toutes les séquences d'échappement à l'exception de `\N{...}` peut dépendre de la plateforme car elles utilisent des codes dépendant de la plate-forme et du codage.

Lire Interpolation en Perl en ligne: https://riptutorial.com/fr/perl/topic/5284/interpolation-en-perl

Chapitre 24: Lecture du contenu d'un fichier dans une variable

Examples

La manière manuelle

```
open my $fh, '<', $filename
    or die "Could not open $filename for reading: $!";
my $contents = do { local $/; <$fh> };
```

Après avoir ouvert le fichier (read `man perlio` si vous voulez lire des codages de fichiers spécifiques au lieu d'octets bruts), l'astuce est dans le bloc `do` : `<$fh>` , le descripteur de fichier dans un opérateur de diamant, renvoie un enregistrement unique . La variable "séparateur d'enregistrements en entrée" `$/` spécifie ce qu'est un "enregistrement" - par défaut, il est défini sur un caractère de nouvelle ligne afin que "un enregistrement" signifie "une seule ligne". Comme `$/` est une variable globale, `local` fait deux choses: il crée une copie locale temporaire de `$/` qui disparaît à la fin du bloc et lui donne la valeur (non) `undef` (la "valeur" que Perl donne aux variables non initialisées). Lorsque le séparateur d'enregistrements en entrée possède cette valeur (non), l'opérateur de diamant retourne le fichier entier. (Il considère le fichier entier comme une seule ligne.)

A l'aide de `do` , vous pouvez même ouvrir manuellement un fichier. Pour la lecture répétée de fichiers,

```
sub readfile { do { local(@ARGV,$/) = $_[0]; <> } }
my $content = readfile($filename);
```

peut être utilisé. Ici, une autre variable globale (`@ARGV`) est localisée pour simuler le même processus utilisé lors du démarrage d'un script perl avec des paramètres. `$/` est toujours `undef` , puisque le tableau devant "mange" tous les arguments entrants. Ensuite, l'opérateur de diamant `<>` fournit à nouveau un enregistrement défini par `$/` (le fichier entier) et retourne du bloc `do` , qui à son tour retourne du sous-fichier.

Le sous n'a pas de gestion explicite des erreurs, ce qui est une mauvaise pratique! Si une erreur survient lors de la lecture du fichier, vous recevrez `undef` comme valeur de retour, par opposition à une chaîne vide provenant d'un fichier vide.

Un autre inconvénient du dernier code est le fait que vous ne pouvez pas utiliser PerlIO pour différents encodages de fichiers - vous obtenez toujours des octets bruts.

Path :: Tiny

Utiliser l'idiome de The Manual Way à plusieurs reprises dans un script devient vite fastidieux. Vous pouvez donc essayer un module.

```
use Path::Tiny;
my $contents = path($filename)->slurp;
```

Vous pouvez passer une option `binmode` si vous avez besoin de contrôler les encodages de fichiers, les fins de ligne, etc. - voir `man perlio` :

```
my $contents = path($filename)->slurp( {binmode => ":encoding(UTF-8)"} );
```

`Path::Tiny` également beaucoup d'autres fonctions pour gérer les fichiers, ce qui en fait un bon choix.

Fichier :: Slurper

C'est un module minimaliste qui ne fait que glisser des fichiers en variables, rien d'autre.

```
use File::Slurper 'read_text';
my $contents = read_text($filename);
```

`read_text()` prend deux paramètres facultatifs pour spécifier le codage du fichier et si les fins de ligne doivent être traduites entre les normes unixish LF ou DOSish CRLF:

```
my $contents = read_text($filename, 'UTF-8', 1);
```

Fichier :: Slurp

Ne l'utilise pas Bien qu'il existe depuis longtemps et soit toujours le module que la plupart des programmeurs suggéreront, il est cassé et il est peu probable qu'il soit corrigé .

Insertion d'un fichier dans une variable de tableau

```
open(my $fh, '<', "/some/path") or die $!;
my @ary = <$fh>;
```

Lorsqu'il est évalué dans un contexte de liste, l'opérateur de diamant renvoie une liste composée de toutes les lignes du fichier (dans ce cas, le résultat est attribué à un contexte de liste de fournitures de tableau). Le terminateur de ligne est conservé et peut être supprimé en chomping:

```
chomp(@ary); #removes line terminators from all the array elements.
```

Fichier Slurp en un trait

Le séparateur d'enregistrement en entrée peut être spécifié avec l'option `-0` (*zéro* , pas de *majuscule O*). Il prend un nombre octal ou hexadécimal comme valeur. Toute valeur supérieure ou égale à `0400` entraînera la perte de fichiers Perl, mais par convention, la valeur utilisée à cette fin est `0777` .

```
perl -0777 -e 'my $file = <>; print length($file)' input.txt
```

En allant plus loin avec le minimalisme, en spécifiant l'option `-n` Perl lit automatiquement chaque ligne (dans notre cas - le fichier entier) dans la variable `$_` .

```
perl -0777 -ne 'print length($_)' input.txt
```

Lire Lecture du contenu d'un fichier dans une variable en ligne:
https://riptutorial.com/fr/perl/topic/1779/lecture-du-contenu-d-un-fichier-dans-une-variable

Chapitre 25: Les meilleures pratiques

Examples

Utiliser Perl :: Critic

Si vous souhaitez commencer à implémenter les meilleures pratiques, pour vous ou votre équipe, Perl :: Critic est le meilleur endroit pour commencer. Le module est basé sur le livre *Perl Best Practices* de Damien Conway et fait un assez bon travail pour implémenter les suggestions qui y sont faites.

> **Note:** *je devrais mentionner (et Conway lui-même dit dans le livre) que ce sont des suggestions. J'ai trouvé que le livre fournit un raisonnement solide dans la plupart des cas, bien que je ne sois certainement pas d'accord avec tous. La chose importante à retenir est que, quelles que soient les pratiques que vous décidez d'adopter, vous restez cohérent. Plus votre code est prévisible, plus il sera facile à maintenir.*

Vous pouvez également essayer Perl :: Critic via votre navigateur sur perlcritic.com .

Installation

```
cpan Perl::Critic
```

Cela installera le jeu de règles de base et un script **perlcritic** pouvant être appelé à partir de la ligne de commande.

Utilisation de base

Le document CPAN pour perlcritic contient une documentation complète. Je ne vais donc que passer en revue les cas d'utilisation les plus courants pour vous lancer. L'utilisation de base consiste simplement à appeler perlcritic sur le fichier:

```
perlcritic -1 /path/to/script.pl
```

perlcritic fonctionne aussi bien sur les scripts que sur les modules. Le **-1** fait référence au niveau de gravité des règles que vous souhaitez exécuter sur le script. Il y a cinq niveaux qui correspondent à combien Perl :: Critic va séparer votre code.

-5 est le plus doux et avertira uniquement des problèmes potentiellement dangereux qui pourraient entraîner des résultats inattendus. **-1** est le plus brutal et se plaindra des choses aussi petites que votre code soit ordonné ou non. D'après mon expérience, garder le code conforme au niveau 3 est suffisant pour rester hors de danger sans devenir trop discret.

Par défaut, les échecs répertorient la raison et la gravité de la règle sur:

```
perlcritic -3 --verbose 8 /path/to/script.pl

Debugging module loaded at line 16, column 1.  You've loaded Data::Dumper, which probably
shouln't be loaded in production.  (Severity: 4)
Private subroutine/method '_sub_name' declared but not used at line 58, column 1.  Eliminate
dead code.  (Severity: 3)
Backtick operator used at line 230, column 37.  Use IPC::Open3 instead.  (Severity: 3)
Backtick operator used at line 327, column 22.  Use IPC::Open3 instead.  (Severity: 3)
```

Affichage des politiques

Vous pouvez rapidement voir quelles règles sont déclenchées et pourquoi en utilisant l'option --verbose de **perlcritic** :

Si vous définissez le niveau sur 8, vous verrez la règle qui a déclenché un avertissement:

```
perlcritic -3 --verbose 8 /path/to/script.pl

[Bangs::ProhibitDebuggingModules] Debugging module loaded at line 16, column 1.  (Severity: 4)
[Subroutines::ProhibitUnusedPrivateSubroutines] Private subroutine/method '_sub_name' declared
but not used at line 58, column 1.  (Severity: 3)
[InputOutput::ProhibitBacktickOperators] Backtick operator used at line 230, column 37.
(Severity: 3)
[InputOutput::ProhibitBacktickOperators] Backtick operator used at line 327, column 22.
(Severity: 3)
```

Alors qu'un niveau de 11 affichera les raisons spécifiques pour lesquelles la règle existe:

```
perlcritic -3 --verbose 11 /path/to/script.pl

Debugging module loaded at line 16, near 'use Data::Dumper;'.
  Bangs::ProhibitDebuggingModules (Severity: 4)
    This policy prohibits loading common debugging modules like the
    Data::Dumper manpage.

    While such modules are incredibly useful during development and
    debugging, they should probably not be loaded in production use. If this
    policy is violated, it probably means you forgot to remove a `use
    Data::Dumper;' line that you had added when you were debugging.
Private subroutine/method '_svn_revisions_differ' declared but not used at line 58, near 'sub
_sub_name {'.
  Subroutines::ProhibitUnusedPrivateSubroutines (Severity: 3)
    By convention Perl authors (like authors in many other languages)
    indicate private methods and variables by inserting a leading underscore
    before the identifier. This policy catches such subroutines which are
    not used in the file which declares them.

    This module defines a 'use' of a subroutine as a subroutine or method
    call to it (other than from inside the subroutine itself), a reference
    to it (i.e. `my $foo = \&_foo'), a `goto' to it outside the subroutine
    itself (i.e. `goto &_foo'), or the use of the subroutine's name as an
    even-numbered argument to `use overload'.
Backtick operator used at line 230, near 'my $filesystem_diff = join q{}, `diff
$trunk_checkout $staging_checkout`;'.
  InputOutput::ProhibitBacktickOperators (Severity: 3)
    Backticks are super-convenient, especially for CGI programs, but I find
```

```
    that they make a lot of noise by filling up STDERR with messages when
    they fail. I think its better to use IPC::Open3 to trap all the output
    and let the application decide what to do with it.

        use IPC::Open3 'open3';
        $SIG{CHLD} = 'IGNORE';

        @output = `some_command`;                       #not ok

        my ($writer, $reader, $err);
        open3($writer, $reader, $err, 'some_command'); #ok;
        @output = <$reader>;  #Output here
        @errors = <$err>;     #Errors here, instead of the console
Backtick operator used at line 327, near 'my $output = `$cmd`;'.
  InputOutput::ProhibitBacktickOperators (Severity: 3)
    Backticks are super-convenient, especially for CGI programs, but I find
    that they make a lot of noise by filling up STDERR with messages when
    they fail. I think its better to use IPC::Open3 to trap all the output
    and let the application decide what to do with it.

        use IPC::Open3 'open3';
        $SIG{CHLD} = 'IGNORE';

        @output = `some_command`;                       #not ok

        my ($writer, $reader, $err);
        open3($writer, $reader, $err, 'some_command'); #ok;
        @output = <$reader>;  #Output here
        @errors = <$err>;     #Errors here, instead of the console
```

Code ignorant

Il y aura des moments où vous ne pouvez pas vous conformer à une politique Perl :: Critic. Dans ces cas, vous pouvez placer des commentaires spéciaux, " **## use Critiques ()** " et " **## non critique** " autour de votre code pour que Perl :: Critic les ignore. Ajoutez simplement les règles que vous souhaitez ignorer entre parenthèses (les multiples peuvent être séparés par une virgule).

```
##no critic qw(InputOutput::ProhibitBacktickOperator)
my $filesystem_diff = join q{}, `diff $trunk_checkout $staging_checkout`;
## use critic
```

Veillez à envelopper le bloc de code entier ou Critic peut ne pas reconnaître l'instruction ignore.

```
## no critic (Subroutines::ProhibitExcessComplexity)
sub no_time_to_refactor_this {
    ...
}
## use critic
```

Notez que certaines stratégies sont exécutées au niveau du document et ne peuvent pas être exemptées de cette manière. Cependant, ils peuvent être désactivés ...

Créer des exceptions permanentes

L'utilisation de ## aucun critique () est bien, mais lorsque vous commencez à adopter des normes de codage, vous voudrez probablement faire des exceptions permanentes à certaines règles. Vous pouvez le faire en créant un fichier de configuration **.perlcriticrc** .

Ce fichier vous permettra de personnaliser non seulement les stratégies exécutées, mais également leur mode d'exécution. Son utilisation est aussi simple que de placer le fichier dans votre répertoire personnel (sous Linux, ne sachant pas si c'est le même endroit sous Windows). Vous pouvez également spécifier le fichier de configuration lors de l'exécution de la commande à l'aide de l'option **--profile** :

```
perlcritic -1 --profile=/path/to/.perlcriticrc /path/to/script.pl
```

Encore une fois, la page CPAN perlcritique contient une liste complète de ces options. Je vais lister quelques exemples de mon propre fichier de configuration:

Appliquer les paramètres de base:

```
#very very harsh
severity = 1
color-severity-medium = bold yellow
color-severity-low = yellow
color-severity-lowest = bold blue
```

Désactiver une règle (notez le tiret devant le nom de la stratégie):

```
# do not require version control numbers
[-Miscellanea::RequireRcsKeywords]

# pod spelling is too over-zealous, disabling
[-Documentation::PodSpelling]
```

Modifier une règle:

```
# do not require checking for print failure ( false positives for printing to stdout, not
filehandle )
[InputOutput::RequireCheckedSyscalls]
    functions = open close

# Allow specific unused subroutines for moose builders
[Subroutines::ProhibitUnusedPrivateSubroutines]
private_name_regex = _(?!build_)\w+
```

Conclusion

Utilisé correctement, Perl :: Critic peut être un outil inestimable pour aider les équipes à maintenir la cohérence et la maintenance de leur codage, quelles que soient les meilleures politiques que vous utilisez.

Lire Les meilleures pratiques en ligne: https://riptutorial.com/fr/perl/topic/5919/les-meilleures-pratiques

Chapitre 26: Les variables

Syntaxe

- ma déclaration # lexicale
- notre # Déclaration globale
- $ foo # Scalar
- @foo # Array
- $ # foo # Array Last-Index
- % foo # Hash
- $ {$ foo} # Dé-référence scalaire
- @ {$ foo} # Array De-Reference
- $ # {$ foo} # Array-DeRef Dernier index
- % {$ foo} # Hash De-Reference
- $ foo [$ index] # Array est indexé
- $ {$ foo} [$ index] # Array De-Reference et obtenir indexés.
- $ foo -> [$ index] # Array De-Reference et obtenir indexé (simplifié)
- $ foo {$ key} # Hash obtient la valeur de la clé
- $ {$ foo} {$ key} # Hash Dereference et récupère la valeur de la clé
- $ foo -> {$ key} # Hash Dereference et obtenir la valeur de la clé (simplifié)
- \ $ x # Référence à Scalar
- \ @x # Référence à un tableau
- \% x # Référence au hachage
- = [] # Référence à un tableau anonyme (Inline)
- = {} # Référence au hachage anonyme (Inline)

Examples

Scalaires

Les scalaires sont le type de données le plus élémentaire de Perl. Ils sont marqués avec le sigil `$` et contiennent une seule valeur de l'un des trois types:

- **un numéro** (`3` , `42` , `3.141` , etc.)
- **une chaîne** (`'hi'` , `"abc"` , etc.)
- **une référence** à une variable (voir d'autres exemples).

```
my $integer = 3;                    # number
my $string = "Hello World";         # string
my $reference = \$string;           # reference to $string
```

Perl convertit à la volée les nombres et les chaînes en fonction des attentes d'un opérateur.

```
my $number = '41';                  # string '41'
my $meaning = $number + 1;          # number  42
```

```
my $sadness = '20 apples';            # string '20 apples'
my $danger = $sadness * 2;            # number '40', raises warning
```

Lors de la conversion d'une chaîne en nombre, Perl prend autant de chiffres que possible sur le recto d'une chaîne - d'où pourquoi `20 apples` sont converties en `20` dans la dernière ligne.

Selon que vous souhaitez traiter le contenu d'un scalaire sous forme de chaîne ou de nombre, vous devez utiliser différents opérateurs. Ne pas les mélanger

```
# String comparison                   # Number comparison
'Potato' eq 'Potato';                 42 == 42;
'Potato' ne 'Pomato';                 42 != 24;
'Camel'  lt 'Potato';                 41 < 42;
'Zombie' gt 'Potato';                 43 > 42;

# String concatenation                # Number summation
'Banana' . 'phone';                   23 + 19;

# String repetition                   # Number multiplication
'nan' x 3;                            6 * 7;
```

Tenter d'utiliser des opérations de chaîne sur des nombres ne déclenchera pas d'avertissement; tenter d'utiliser des opérations numériques sur des chaînes non numériques sera. Sachez que certaines chaînes non numériques telles que `'inf'`, `'nan'`, `'0 but true'` comptent comme des nombres.

Tableaux

Les tableaux stockent une séquence ordonnée de valeurs. Vous pouvez accéder au contenu par index ou les parcourir. Les valeurs resteront dans l'ordre dans lequel vous les avez remplies.

```
my @numbers_to_ten = (1,2,3,4,5,6,7,8,9,10); # More conveniently: (1..10)
my @chars_of_hello = ('h','e','l','l','o');
my @word_list = ('Hello','World');

# Note the sigil: access an @array item with $array[index]
my $second_char_of_hello = $chars_of_hello[1]; # 'e'

# Use negative indices to count from the end (with -1 being last)
my $last_char_of_hello = $chars_of_hello[-1];

# Assign an array to a scalar to get the length of the array
my $length_of_array = @chars_of_hello; # 5

# You can use $# to get the last index of an array, and confuse Stack Overflow
my $last_index_of_array = $#chars_of_hello; # 4

# You can also access multiple elements of an array at the same time
# This is called "array slice"
# Since this returns multiple values, the sigil to use here on the RHS is @
my @some_chars_of_hello = @chars_of_hello[1..3]; # ('H', 'e', 'l')
my @out_of_order_chars = @chars_of_hello[1,4,2]; # ('e', 'o', 'l')

# In Python you can say array[1:-1] to get all elements but first and last
# Not so in Perl: (1..-1) is an empty list. Use $# instead
```

```
my @empty_list = @chars_of_hello[1..-1];                            # ()
my @inner_chars_of_hello = @chars_of_hello[1..$#chars_of_hello-1]; # ('e','l','l')

# Access beyond the end of the array yields undef, not an error
my $undef = $chars_of_hello[6]; # undef
```

Les tableaux sont mutables:

```
use utf8; # necessary because this snippet is utf-8
$chars_of_hello[1] = 'u';              #     ('h','u','l','l','o')
push @chars_of_hello, ('!', '!');      #     ('h','u','l','l','o','!','!')
pop @chars_of_hello;                   #     ('h','u','l','l','o','!')
shift @chars_of_hello;                 #         ('u','l','l','o','!')
unshift @chars_of_hello, ('¡', 'H');   # ('¡','H','u','l','l','o','!')
@chars_of_hello[2..5] = ('O','L','A'); # ('¡','H','O','L','A',undef,'!') whoops!
delete $chars_of_hello[-2];            # ('¡','H','O','L','A',      '!')

# Setting elements beyond the end of an array does not result in an error
# The array is extended with undef's as necessary. This is "autovivification."
my @array;           # ()
my @array[3] = 'x';  # (undef, undef, undef, 'x')
```

Enfin, vous pouvez parcourir le contenu d'un tableau:

```
use v5.10; # necessary for 'say'
for my $number (@numbers_to_ten) {
  say $number ** 2;
}
```

Lorsqu'ils sont utilisés comme booléens, les tableaux sont vrais s'ils ne sont pas vides.

Hash

Les hachages peuvent être compris comme des tables de correspondance. Vous pouvez accéder à son contenu en spécifiant une clé pour chacun d'entre eux. Les clés doivent être des chaînes. Si ce n'est pas le cas, ils seront convertis en chaînes.

Si vous donnez simplement au hash une clé connue, cela vous servira sa valeur.

```
# Elements are in (key, value, key, value) sequence
my %inhabitants_of = ("London", 8674000, "Paris", 2244000);

# You can save some typing and gain in clarity by using the "fat comma"
# syntactical sugar. It behaves like a comma and quotes what's on the left.
my %translations_of_hello = (spanish => 'Hola', german => 'Hallo', swedish => 'Hej');
```

Dans l'exemple suivant, notez les crochets et sigil: vous accédez à un élément de `%hash` utilisant `$hash{key}` car la *valeur que* vous voulez est un scalaire. Certains considèrent qu'il est recommandé de citer la clé alors que d'autres trouvent ce style visuellement bruyant. La citation n'est requise que pour les clés qui pourraient être confondues avec des expressions telles que `$hash{'some-key'}`

```
my $greeting = $translations_of_hello{'spanish'};
```

Alors que Perl essaiera par défaut d'utiliser des mots nus comme des chaînes, `+` modificateur peut également être utilisé pour indiquer à Perl que la clé ne doit pas être interpolée mais exécutée avec le résultat de l'exécution utilisé comme clé:

```
my %employee = ( name => 'John Doe', shift => 'night' );
# this example will print 'night'
print $employee{shift};

# but this one will execute [shift][1], extracting first element from @_,
# and use result as a key
print $employee{+shift};
```

Comme avec les tableaux, vous pouvez accéder à plusieurs éléments de hachage en même temps. Ceci est appelé une *tranche de hachage* . La valeur résultante est une liste, utilisez donc le sigil `@` :

```
my @words = @translations_of_hello{'spanish', 'german'}; # ('Hola', 'Hallo')
```

Itérer sur les clés d'un hachage avec les `keys` `keys` renverra les éléments dans un ordre aléatoire. Combinez avec le `sort` si vous le souhaitez.

```
for my $lang (sort keys %translations_of_hello) {
  say $translations_of_hello{$lang};
}
```

Si vous n'avez pas besoin des clés comme dans l'exemple précédent, `values` renvoie directement les valeurs de hachage:

```
for my $translation (values %translations_of_hello) {
  say $translation;
}
```

Vous pouvez également utiliser une boucle while avec `each` pour parcourir le hachage. De cette façon, vous obtiendrez à la fois la clé et la valeur, sans recherche de valeur séparée. Son utilisation est cependant déconseillée, `each` pouvant se briser de manière floue.

```
# DISCOURAGED
while (my ($lang, $translation) = each %translations_of_hello) {
  say $translation;
}
```

L'accès aux éléments non définis renvoie undef, pas une erreur:

```
my $italian = $translations_of_hello{'italian'}; # undef
```

`map` aplatissement des `map` et des listes peut être utilisé pour créer des hachages à partir de tableaux. C'est un moyen populaire de créer un ensemble de valeurs, par exemple pour vérifier rapidement si une valeur est dans `@elems` . Cette opération prend généralement O (n) time (c'est-à-dire proportionnelle au nombre d'éléments) mais peut être effectuée à temps constant (O (1)) en

transformant la liste en hash:

```
@elems = qw(x y x z t);
my %set = map { $_ => 1 } @elems;   # (x, 1, y, 1, t, 1)
my $y_membership = $set{'y'};       # 1
my $w_membership = $set{'w'};       # undef
```

Cela nécessite des explications. Le contenu de `@elems` est lu dans une liste qui est traitée par `map` . `map` accepte un bloc de code appelé pour chaque valeur de sa liste d'entrée; la valeur de l'élément est disponible pour utilisation dans `$_` . Notre bloc de code renvoie *deux* éléments de liste pour chaque élément d'entrée: `$_` , l'élément d'entrée et `1` , juste une valeur. Une fois que vous avez comptabilisé l'aplatissement des listes, le résultat est que la `map { $_ => 1 } @elems` transforme `qw(xyxzt)` en `(x => 1, y => 1, x => 1, z => 1, t => 1)` .

Lorsque ces éléments sont affectés au hachage, les éléments impairs deviennent des clés de hachage et même les éléments deviennent des valeurs de hachage. Lorsqu'une clé est spécifiée plusieurs fois dans une liste pour être affectée à un hachage, la *dernière* valeur gagne. Cela élimine efficacement les doublons.

Un moyen plus rapide de transformer une liste en une table de hachage utilise l'affectation à une tranche de hachage. Il utilise l'opérateur `x` pour multiplier la liste à un seul élément `(1)` par la taille de `@elems` , il y a donc une valeur `1` pour chacune des clés de la tranche à gauche:

```
@elems = qw(x y x z t);
my %set;
@set{@elems} = (1) x @elems;
```

L'application de hachage suivante exploite également le fait que les hachages et les listes peuvent souvent être utilisés indifféremment pour implémenter des arguments de fonction nommés:

```
sub hash_args {
  my %args = @_;
  my %defaults = (foo => 1, bar => 0);
  my %overrides = (__unsafe => 0);
  my %settings = (%defaults, %args, %overrides);
}

# This function can then be called like this:
hash_args(foo => 5, bar => 3); # (foo => 5, bar => 3, __unsafe ==> 0)
hash_args();                   # (foo => 1, bar => 0, __unsafe ==> 0)
hash_args(__unsafe => 1)       # (foo => 1, bar => 0, __unsafe ==> 0)
```

Lorsqu'ils sont utilisés comme des booléens, les hachages sont vrais s'ils ne sont pas vides.

Références scalaires

Une **référence** est une variable scalaire (une préfixée par `$`) qui «renvoie» à d'autres données.

```
my $value     = "Hello";
my $reference = \$value;
print $value;     # => Hello
```

```
print $reference; # => SCALAR(0x2683310)
```

Pour obtenir les données **référencées** , vous les **référencez** .

```
say ${$reference};                  # Explicit prefix syntax
say $$reference;                    # The braces can be left out (confusing)
```

5.24.0

Nouvelle syntaxe de déréférencement postfix, disponible par défaut à partir de la v5.24

```
use v5.24;
say $reference->$*; # New postfix notation
```

Cette "valeur dé-référencée" peut alors être modifiée comme si c'était la variable d'origine.

```
${$reference} =~ s/Hello/World/;
print ${$reference};  # => World
print $value;         # => World
```

Une référence est toujours **véridique** - même si la valeur à laquelle elle fait référence est falsifiée (comme `0` ou `""`).

Vous voudrez peut-être une référence scalaire si:

- Vous voulez passer une chaîne à une fonction et la modifier pour vous sans qu'il s'agisse d'une valeur de retour.
- Vous souhaitez éviter explicitement que Perl copie implicitement le contenu d'une chaîne de caractères volumineuse à un moment donné de votre fonction (particulièrement sur les versions antérieures de Perls sans les chaînes de copie).
- Vous souhaitez désambiguïser des valeurs de type chaîne avec une signification spécifique, à partir de chaînes qui véhiculent du contenu, par exemple:
    - Désambiguïser un nom de fichier du contenu du fichier
    - Désambiguïser le contenu renvoyé à partir d'une chaîne d'erreur renvoyée
- Vous souhaitez implémenter un modèle d'objet fin et léger, dans lequel les objets transmis au code d'appel ne comportent pas de métadonnées visibles par l'utilisateur:

    ```
    our %objects;
    my $next_id = 0;
    sub new {
       my $object_id = $next_id++;
       $objects{ $object_id } = { ... }; # Assign data for object
       my $ref = \$object_id;
       return bless( $ref, "MyClass" );
    }
    ```

Références de tableau

Les références de tableaux sont des scalaires (`$`) qui font référence à des tableaux.

```
my @array = ("Hello"); # Creating array, assigning value from a list
my $array_reference = \@array;
```

Ceux-ci peuvent être créés plus court comme suit:

```
my $other_array_reference = ["Hello"];
```

La modification / utilisation de références de tableau nécessite leur suppression.

```
my @contents = @{ $array_reference };               # Prefix notation
my @contents = @$array_reference;                   # Braces can be left out
```

5.24.0

Nouvelle syntaxe de déréférencement postfix, disponible par défaut à partir de la v5.24

```
use v5.24;
my @contents = $array_reference->@*; # New postfix notation
```

Lorsque vous accédez au contenu d'une table RAF par index, vous pouvez utiliser le sucre syntaxique `->` .

```
my @array = qw(one two three);      my $arrayref = [ qw(one two three) ]
my $one = $array[0];                my $one = $arrayref->[0];
```

Contrairement aux tableaux, les tableaux peuvent être imbriqués:

```
my @array = ( (1, 0), (0, 1) )  # ONE array of FOUR elements: (1, 0, 0, 1)
my @matrix = ( [1, 0], [0, 1] ) # an array of two arrayrefs
my $matrix = [ [0, 1], [1, 0] ] # an arrayref of arrayrefs
# There is no namespace conflict between scalars, arrays and hashes
# so @matrix and $matrix _both_ exist at this point and hold different values.

my @diagonal_1 = ($matrix[0]->[1], $matrix[1]->[0])     # uses @matrix
my @diagonal_2 = ($matrix->[0]->[1], $matrix->[1]->[0]) # uses $matrix
# Since chained []- and {}-access can only happen on references, you can
# omit some of those arrows.
my $corner_1 = $matrix[0][1];   # uses @matrix;
my $corner_2 = $matrix->[0][1]; # uses $matrix;
```

Lorsqu'elles sont utilisées comme booléennes, les références sont toujours vraies.

Références de hachage

Les références de hachage sont des scalaires qui contiennent un pointeur sur l'emplacement de la mémoire contenant les données d'un hachage. Étant donné que le scalaire pointe directement sur

le hachage lui-même, lorsqu'il est transmis à un sous-programme, les modifications apportées au hachage ne sont pas locales au sous-programme comme avec un hachage normal, mais sont globales.

Tout d'abord, examinons ce qui se passe lorsque vous passez un hachage normal à un sous-programme et que vous le modifiez ici:

```
use strict;
use warnings;
use Data::Dumper;

sub modify
{
    my %hash = @_;

    $hash{new_value} = 2;

    print Dumper("Within the subroutine");
    print Dumper(\%hash);

    return;
}

my %example_hash = (
    old_value   => 1,
);

modify(%example_hash);

print Dumper("After exiting the subroutine");
print Dumper(\%example_hash);
```

Qui aboutit à:

```
$VAR1 = 'Within the subroutine';
$VAR1 = {
          'new_value' => 2,
          'old_value' => 1
        };
$VAR1 = 'After exiting the subroutine';
$VAR1 = {
          'old_value' => 1
        };
```

Notez qu'après avoir quitté le sous-programme, le hachage reste inchangé; toutes les modifications apportées étaient locales au sous-programme de modification, car nous avons transmis une copie du hachage, et non le hachage lui-même.

En comparaison, lorsque vous transmettez un hashref, vous transmettez l'adresse au hachage d'origine, de sorte que toute modification apportée dans le sous-programme sera effectuée sur le hachage d'origine:

```
use strict;
use warnings;
use Data::Dumper;
```

```
sub modify
{
    my $hashref = shift;

    # De-reference the hash to add a new value
    $hashref->{new_value} = 2;

    print Dumper("Within the subroutine");
    print Dumper($hashref);

    return;
}

# Create a hashref
my $example_ref = {
    old_value   => 1,
};

# Pass a hashref to a subroutine
modify($example_ref);

print Dumper("After exiting the subroutine");
print Dumper($example_ref);
```

Cela se traduira par:

```
$VAR1 = 'Within the subroutine';
$VAR1 = {
          'new_value' => 2,
          'old_value' => 1
        };
$VAR1 = 'After exiting the subroutine';
$VAR1 = {
          'new_value' => 2,
          'old_value' => 1
        };
```

Typeglobs, réfs de typeglob, descripteurs de fichiers et constantes

Un typeglob `*foo` contient des références au contenu des variables *globales* portant ce nom: `$foo` , `@foo` , `$foo` , `&foo` , etc. Vous pouvez y accéder comme un hachage et affecter directement les tables de symboles (evil!).

```
use v5.10; # necessary for say
our $foo = "foo";
our $bar;
say ref *foo{SCALAR};     # SCALAR
say ${ *foo{SCALAR} };    # bar
*bar = *foo;
say $bar;                 # bar
$bar = 'egg';
say $foo;                 # egg
```

Les typesglobs sont plus couramment utilisés pour traiter les fichiers. `open` , par exemple, crée une référence à un typeglob lorsqu'on lui demande de créer un descripteur de fichier non global:

```
use v5.10; # necessary for say
open(my $log, '> utf-8', '/tmp/log') or die $!; # open for writing with encoding
say $log 'Log opened';

# You can dereference this globref, but it's not very useful.
say ref $log;                    # GLOB
say (*{$log}->{IO} // 'undef'); # undef

close $log or die $!;
```

Typeglobs peut également être utilisé pour créer des variables en lecture seule globales, bien que l'`use constant` soit plus répandue.

```
# Global constant creation
*TRUE = \('1');
our $TRUE;
say $TRUE;  # 1
$TRUE = ''; # dies, "Modification of a read-only value attempted"

# use constant instead defines a parameterless function, therefore it's not global,
# can be used without sigils, can be imported, but does not interpolate easily.
use constant (FALSE => 0);
say FALSE;        # 0
say &FALSE;       # 0
say "${\FALSE}";  # 0 (ugh)
say *FALSE{CODE}; # CODE(0xMA1DBABE)

# Of course, neither is truly constant when you can manipulate the symbol table...
*TRUE = \('');
use constant (EVIL => 1);
*FALSE = *EVIL;
```

Sigils

Perl a plusieurs sigils:

```
$scalar = 1; # individual value
@array = ( 1, 2, 3, 4, 5 ); # sequence of values
%hash = ('it', 'ciao', 'en', 'hello', 'fr', 'salut'); # unordered key-value pairs
&function('arguments'); # subroutine
*typeglob; # symbol table entry
```

Celles-ci ressemblent à des sceaux, mais ne sont pas:

```
\@array; # \ returns the reference of what's on the right (so, a reference to @array)
$#array; # this is the index of the last element of @array
```

Vous pouvez utiliser des accolades après le sigil si vous le souhaitez. Cela améliore parfois la lisibilité.

```
say ${value} = 5;
```

Bien que vous utilisiez des sigils différents pour définir des variables de différents types, la même

variable est accessible de différentes manières en fonction des sigils que vous utilisez.

```
%hash;            # we use % because we are looking at an entire hash
$hash{it};        # we want a single value, however, that's singular, so we use $
$array[0];        # likewise for an array. notice the change in brackets.
@array[0,3];      # we want multiple values of an array, so we instead use @
@hash{'it','en'}; # similarly for hashes (this gives the values: 'ciao', 'hello')
%hash{'it','fr'}; # we want an hash with just some of the keys, so we use %
                  # (this gives key-value pairs: 'it', 'ciao', 'fr', 'salut')
```

Ceci est particulièrement vrai pour les références. Pour utiliser une valeur référencée, vous pouvez combiner des sigils ensemble.

```
my @array = 1..5;                     # This is an array
my $reference_to_an_array = \@array; # A reference to an array is a singular value
push @array, 6;                       # push expects an array
push @$reference_to_an_array, 7;      # the @ sigil means what's on the right is an array
                                      # and what's on the right is $reference_to_an_array
                                      # hence: first a @, then a $
```

Voici une façon peut-être moins déroutante d'y penser. Comme nous l'avons vu précédemment, vous pouvez utiliser des accolades pour envelopper ce qui se trouve à droite d'un sceau. Vous pouvez donc penser à `@{}` comme quelque chose qui prend une référence de tableau et vous donne le tableau référencé.

```
# pop does not like array references
pop $reference_to_an_array; # ERROR in Perl 5.20+
# but if we use @{}, then...
pop @{ $reference_to_an_array }; # this works!
```

En fait, `@{}` accepte en réalité une expression:

```
my $values = undef;
say pop @{ $values };       # ERROR: can't use undef as an array reference
say pop @{ $values // [5] } # undef // [5] gives [5], so this prints 5
```

... et le même truc fonctionne aussi pour les autres sceaux.

```
# This is not an example of good Perl. It is merely a demonstration of this language feature
my $hashref = undef;
for my $key ( %{ $hashref // {} } ) {
  "This doesn't crash";
}
```

... mais si "l'argument" d'un sceau est simple, vous pouvez laisser les accolades.

```
say $$scalar_reference;
say pop @$array_reference;
for keys (%$hash_reference) { ... };
```

Les choses peuvent devenir excessivement extravagantes. Cela fonctionne, mais s'il vous plaît Perl responsable.

```
my %hash = (it => 'ciao', en => 'hi', fr => 'salut');
my $reference = \%hash;
my $reference_to_a_reference = \$reference;

my $italian = $hash{it};                              # Direct access
my @greets = @$reference{'it', 'en'};                 # Dereference, then access as array
my %subhash = %$$reference_to_a_reference{'en', 'fr'} # Dereference ×2 then access as hash
```

Pour une utilisation normale, vous pouvez simplement utiliser des noms de sous-programmes sans sigil. (Variables sans Sigil sont généralement appelés « barewords ».) Le `&` Sigil est utile que dans un nombre limité de cas.

- Faire une référence à une sous-routine:

  ```
  sub many_bars { 'bar' x $_[0] }
  my $reference = \&many_bars;
  say $reference->(3); # barbarbar
  ```

- Appel d'une fonction en ignorant son prototype.

- Combiné avec goto, comme un appel de fonction légèrement étrange dont la trame d'appel en cours est remplacée par l'appelant. Pensez à l'appel API linux `exec()` , mais pour les fonctions.

Lire Les variables en ligne: https://riptutorial.com/fr/perl/topic/1566/les-variables

Chapitre 27: Optimisation de l'utilisation de la mémoire

Examples

Lecture de fichiers: foreach vs. while

Lors de la lecture d' un fichier potentiellement important, un `while` boucle a un avantage de mémoire importante sur `foreach` . Ce qui suit lira l'enregistrement de fichier par enregistrement (par défaut, "record" signifie "une ligne", comme spécifié par `$/`), en attribuant chacun à `$_` tel qu'il est lu:

```
while(<$fh>) {
    print;
}
```

L' *opérateur de diamant* fait de la magie ici pour s'assurer que la boucle ne se termine qu'à la fin du fichier et non par exemple sur les lignes contenant uniquement un caractère "0".

La boucle suivante semble fonctionner de la même manière, mais elle évalue l'opérateur de diamant dans un contexte de liste, ce qui entraîne la lecture complète du fichier:

```
foreach(<$fh>) {
    print;
}
```

Si vous utilisez de toute façon un enregistrement à la fois, cela peut entraîner un énorme gaspillage de mémoire et doit donc être évité.

Traitement de longues listes

Si vous avez déjà une liste en mémoire, le moyen le plus simple et le plus simple de le traiter est une simple boucle `foreach` :

```
foreach my $item (@items) {
    ...
}
```

Cela convient bien, par exemple, au traitement courant de `$item` et à son écriture dans un fichier sans conserver les données. Toutefois, si vous construisez une autre structure de données à partir des éléments, une `while` boucle est plus efficace mémoire:

```
my @result;
while(@items) {
    my $item = shift @items;
    push @result, process_item($item);
```

```
}
```

À moins qu'une référence à `$item` se retrouve directement dans votre liste de résultats, les éléments que vous avez décalés du tableau `@items` peuvent être libérés et la mémoire réutilisée par l'interpréteur lorsque vous entrez dans la prochaine itération de la boucle.

Lire Optimisation de l'utilisation de la mémoire en ligne:
https://riptutorial.com/fr/perl/topic/6327/optimisation-de-l-utilisation-de-la-memoire

Chapitre 28: Packages et modules

Syntaxe

- require Module :: Name; # Requis par nom de @INC
- nécessite "path / to / file.pm"; # Requis par chemin relatif depuis @INC
- utiliser Module :: Name; # require et import par défaut à BEGIN
- utilisez Module :: Name (); # requis et pas d'importation à BEGIN
- utiliser Module :: Name (@ARGS); # nécessite et importe avec args à BEGIN
- utiliser Module :: Name VERSION; # demande, contrôle de version et importation par défaut à BEGIN
- utiliser Module :: Name VERSION (); # demande, contrôle de la version et pas d'importation à BEGIN
- utilisez Module :: Name VERSION (@ARGS); # requis, contrôle de version, import avec args à BEGIN
- faire "path / to / file.pl"; # charger et évaluer le fichier donné

Examples

Exécuter le contenu d'un autre fichier

```
do './config.pl';
```

Cela lira le contenu du fichier config.pl et l'exécutera. (Voir aussi: `perldoc -f do` .)

NB: Evitez de `do` sauf si vous jouez au golf ou quelque chose comme il n'y a pas de vérification d'erreur. Pour inclure des modules de bibliothèque, utilisez `require` ou `use` .

Chargement d'un module à l'exécution

```
require Exporter;
```

Cela garantira que le module `Exporter` est chargé à l'exécution s'il n'a pas déjà été importé. (Voir aussi: `perldoc -f require` .)

NB: La plupart des utilisateurs doivent `use` modules plutôt que `require` les `require` . Contrairement à `use` , `require` n'appelle pas la méthode d'importation du module et s'exécute à l'exécution, pas pendant la compilation.

Cette façon de charger les modules est utile si vous ne pouvez pas décider des modules dont vous avez besoin avant l'exécution, comme avec un système de plug-in:

```
package My::Module;
my @plugins = qw( One Two );
foreach my $plugin (@plugins) {
```

```
    my $module = __PACKAGE__ . "::Plugins::$plugin";
    $module =~ s!::!/!g;
    require "$module.pm";
}
```

Cela essaierait de charger `My::Package::Plugins::One` et `My::Package::Plugins::Two` . `@plugins` devrait bien sûr provenir d'une entrée utilisateur ou d'un fichier de configuration pour que cela ait un sens. Notez l'opérateur de substitution `s!::!/!g` qui remplace chaque paire de deux-points par une barre oblique. En effet , vous pouvez charger des modules en utilisant la syntaxe du nom du module familier de l' `use` que si le nom du module est un bareword. Si vous transmettez une chaîne ou une variable, elle doit contenir un nom de fichier.

En utilisant un module

```
use Cwd;
```

Cela importera le module `Cwd` au moment de la compilation et importera ses symboles par défaut, c'est-à-dire que certaines des variables et fonctions du module seront disponibles pour le code qui l'utilise. (Voir aussi: `perldoc -f use` .)

Généralement, cela fera la bonne chose. Cependant, vous voudrez parfois contrôler les symboles importés. Ajoutez une liste de symboles après le nom du module à exporter:

```
use Cwd 'abs_path';
```

Si vous faites cela, seuls les symboles que vous spécifiez seront importés (c.-à-d. Que le jeu par défaut ne sera pas importé).

Lors de l'importation de plusieurs symboles, il est idiomatique d'utiliser la construction de liste `qw()` :

```
use Cwd qw(abs_path realpath);
```

Certains modules exportent un sous-ensemble de leurs symboles, mais on peut leur demander d’exporter tout avec `:all` :

```
use Benchmark ':all';
```

(Notez que tous les modules ne reconnaissent pas ou n’utilisent pas la balise `:all`).

Utiliser un module dans un répertoire

```
use lib 'includes';
use MySuperCoolModule;
```

`use lib 'includes';` ajoute le répertoire relatif `includes/` comme autre chemin de recherche de module dans `@INC` . Supposons donc que vous avez un fichier de module `MySyperCoolModule.pm`

intérieur de `MySyperCoolModule.pm` `includes/` , qui contient:

```
package MySuperCoolModule;
```

Si vous le souhaitez, vous pouvez regrouper autant de modules que vous le souhaitez dans un même répertoire et les rendre accessibles avec une seule instruction `use lib` .

À ce stade, l'utilisation des sous-routines du module nécessitera de préfixer le nom du sous-programme avec le nom du package:

```
MySuperCoolModule::SuperCoolSub_1("Super Cool String");
```

Pour pouvoir utiliser les sous-programmes sans préfixe, vous devez exporter les noms des sous-programmes afin qu'ils soient reconnus par le programme qui les appelle. L'exportation peut être configurée pour être automatique, donc:

```
package MySuperCoolModule;
use base 'Exporter';
our @EXPORT = ('SuperCoolSub_1', 'SuperCoolSub_2');
```

Ensuite, dans le fichier qui `use` le module, ces sous-routines seront automatiquement disponibles:

```
use MySuperCoolModule;
SuperCoolSub_1("Super Cool String");
```

Ou vous pouvez configurer le module pour exporter conditionnellement des sous-routines, donc:

```
package MySuperCoolModule;
use base 'Exporter';
our @EXPORT_OK = ('SuperCoolSub_1', 'SuperCoolSub_2');
```

Dans ce cas, vous devez explicitement demander l'exportation des sous-routines souhaitées dans le script `use` le module:

```
use MySuperCoolModule 'SuperCoolSub_1';
SuperCoolSub_1("Super Cool String");
```

CPAN.pm

`CPAN.pm` est un module Perl qui permet d'interroger et d'installer des modules à partir de sites CPAN.

Il supporte le mode interactif invoqué avec

```
cpan
```

ou

```
perl -MCPAN -e shell
```

Interroger les modules

De nom:

```
cpan> m MooseX::YAML
```

Par une expression régulière contre le nom du module:

```
cpan> m /^XML::/
```

Remarque: pour activer un pager ou rediriger vers un fichier `|` *ou* `>` *redirection de shell (les espaces sont obligatoires autour des* `|` *et* `>` *), par exemple:* `m /^XML::/ | less`

Par distribution:

```
cpan> d LMC/Net-Squid-Auth-Engine-0.04.tar.gz
```

Installation de modules

De nom:

```
cpan> install MooseX::YAML
```

Par distribution:

```
cpan> install LMC/Net-Squid-Auth-Engine-0.04.tar.gz
```

Liste tous les modules installés

En ligne de commande:

```
cpan -l
```

À partir d'un script Perl:

```
use ExtUtils::Installed;
my $inst = ExtUtils::Installed->new();
my @modules = $inst->modules();
```

Lire Packages et modules en ligne: https://riptutorial.com/fr/perl/topic/451/packages-et-modules

Chapitre 29: Perl one-liners

Examples

Exécuter du code Perl en ligne de commande

De simples lignes simples peuvent être spécifiées comme arguments de ligne de commande pour perl en utilisant le commutateur `-e` (pensez "execute"):

```
perl -e'print "Hello, World!\n"'
```

En raison de règles de citation de Windows, vous ne pouvez pas utiliser de chaînes entre guillemets simples, mais vous devez utiliser l'une de ces variantes:

```
perl -e"print qq(Hello, World!\n)"
perl -e"print \"Hello, World!\n\""
```

Notez que pour éviter de casser l'ancien code, seule la syntaxe disponible jusqu'à Perl 5.8.x peut être utilisée avec `-e` . Pour utiliser quelque chose de plus récent, votre version de perl peut prendre en charge, utilisez plutôt `-E` . Par exemple, utiliser `say` disponible à partir de 5.10.0 sur Unicode 6.0 à partir de> = v5.14.0 (utilise également `-CO` pour s'assurer que `STDOUT` imprime UTF-8):

5.14.0

```
perl -CO -E'say "\N{PILE OF POO}"'
```

Utilisation de chaînes entre guillemets doubles dans les guichets uniques de Windows

Windows utilise uniquement des guillemets doubles pour envelopper les paramètres de ligne de commande. Pour utiliser des guillemets doubles dans perl one-liner (c'est-à-dire pour imprimer une chaîne avec une variable interpolée), vous devez leur échapper avec des barres obliques inverses:

```
perl -e "my $greeting = 'Hello'; print \"$greeting, world!\n\""
```

Pour améliorer la lisibilité, vous pouvez utiliser un opérateur `qq()` :

```
perl -e "my $greeting = 'Hello'; print qq($greeting, world!\n)"
```

Imprimer des lignes correspondant à un motif (PCRE grep)

```
perl -ne'print if /foo/' file.txt
```

Insensible à la casse:

```
perl -ne'print if /foo/i' file.txt
```

Remplacer une sous-chaîne par une autre (PCRE sed)

```
perl -pe"s/foo/bar/g" file.txt
```

Ou en place:

```
perl -i -pe's/foo/bar/g' file.txt
```

Sous Windows:

```
perl -i.bak -pe"s/foo/bar/g" file.txt
```

Imprimer seulement certains champs

```
perl -lane'print "$F[0] $F[-1]"' data.txt
# prints the first and the last fields of a space delimited record
```

Exemple CSV:

```
perl -F, -lane'print "$F[0] $F[-1]"' data.csv
```

Imprimer les lignes 5 à 10

```
perl -ne'print if 5..10' file.txt
```

Modifier le fichier sur place

Sans copie de sauvegarde (non prise en charge sous Windows)

```
perl -i -pe's/foo/bar/g' file.txt
```

Avec une copie de sauvegarde `file.txt.bak`

```
perl -i.bak -pe's/foo/bar/g' file.txt
```

Avec une copie de sauvegarde `old_file.txt.orig` dans le sous-répertoire `backup` (à condition que ce dernier existe):

```
perl -i'backup/old_*.orig' -pe's/foo/bar/g' file.txt
```

Lecture du fichier entier sous forme de chaîne

```
perl -0777 -ne'print "The whole file as a string: --->$_<---\n"'
```

Note: Le `-0777` est juste une convention. N'importe quel `-0400` et au-dessus serait le même.

Télécharger le fichier dans mojolicious

```
perl -Mojo -E 'p("http://localhost:3000" => form => {Input_Type => "XML", Input_File => {file
=> "d:/xml/test.xml"}})'
```

Le fichier `d:/xml/test.xml` sera chargé sur le serveur qui écoute les connexions sur `localhost:3000` (Source)

Dans cet exemple:

`-Mmodule` exécute le `use module;` avant d'exécuter votre programme
`-E commandline` permet d'entrer une ligne de programme
Si vous n'avez pas de module `ojo` vous pouvez utiliser la commande `cpanm ojo` pour l'installer

Pour en savoir plus sur l'exécution de perl, utilisez la commande `perldoc perlrun` ou lisez ici

Lire Perl one-liners en ligne: https://riptutorial.com/fr/perl/topic/3696/perl-one-liners

Chapitre 30: Perl orienté objet

Examples

Créer des objets

Contrairement à de nombreux autres langages, Perl ne dispose pas de constructeurs qui allouent de la mémoire pour vos objets. Au lieu de cela, il convient d'écrire une méthode de classe qui à la fois crée une structure de données et la remplit avec des données (vous pouvez le connaître comme modèle de conception de méthode d'usine).

```
package Point;
use strict;

sub new {
    my ($class, $x, $y) = @_;
    my $self = { x => $x, y => $y }; # store object data in a hash
    bless $self, $class;             # bind the hash to the class
    return $self;
}
```

Cette méthode peut être utilisée comme suit:

```
my $point = Point->new(1, 2.5);
```

Lorsque l'opérateur flèche `->` est utilisé avec des méthodes, son opérande gauche est ajouté à la liste d'arguments indiquée. Donc, `@_` in `new` contiendra des valeurs `('Point', 1, 2.5)` .

Il n'y a rien de spécial dans le nom `new` . Vous pouvez appeler les méthodes d'usine comme vous préférez.

Il n'y a rien de spécial dans les hashes. Vous pouvez faire la même chose de la manière suivante:

```
package Point;
use strict;

sub new {
    my ($class, @coord) = @_;
    my $self = \@coord;
    bless $self, $class;
    return $self;
}
```

En général, toute référence peut être un objet, même une référence scalaire. Mais le plus souvent, les hachages constituent le moyen le plus pratique de représenter des données d'objet.

Définir des classes

En général, les classes en Perl ne sont que des packages. Ils peuvent contenir des données et

des méthodes, comme les paquets habituels.

```
package Point;
use strict;

my $CANVAS_SIZE = [1000, 1000];

sub new {
    ...
}

sub polar_coordinates {
    ...
}

1;
```

Il est important de noter que les variables déclarées dans un package sont des variables de classe, pas des variables d'objet (instance). La modification d'une variable au niveau du package affecte tous les objets de la classe. Comment stocker des données spécifiques à un objet, voir dans «Création d'objets».

Ce qui rend les paquets de classe spécifiques, c'est l'opérateur flèche `->` . Il peut être utilisé après un mot nu:

```
Point->new(...);
```

ou après une variable scalaire (contenant généralement une référence):

```
my @polar = $point->polar_coordinates;
```

Ce qui est à gauche de la flèche est ajouté à la liste des arguments de la méthode. Par exemple, après un appel

```
Point->new(1, 2);
```

array `@_` in `new` contiendra trois arguments: `('Point', 1, 2)` .

Les packages représentant des classes doivent prendre en compte cette convention et s’attendre à ce que toutes leurs méthodes aient un argument supplémentaire.

Résolution d'héritage et de méthodes

Pour faire une classe une sous-classe d'une autre classe, utilisez le pragma `parent` :

```
package Point;
use strict;
...
1;

package Point2D;
```

```
use strict;
use parent qw(Point);
...
1;

package Point3D;
use strict;
use parent qw(Point);
...
1;
```

Perl permet l'héritage multiple:

```
package Point2D;
use strict;
use parent qw(Point PlanarObject);
...
1;
```

Héritage est tout au sujet de la résolution quelle méthode doit être appelée dans une situation particulière. Étant donné que pure Perl ne prescrit aucune règle concernant la structure de données utilisée pour stocker les données d'objet, l'héritage n'a rien à voir avec cela.

Considérons la hiérarchie de classes suivante:

```
package GeometryObject;
use strict;

sub transpose { ...}

1;

package Point;
use strict;
use parent qw(GeometryObject);

sub new { ... };

1;

package PlanarObject;
use strict;
use parent qw(GeometryObject);

sub transpose { ... }

1;

package Point2D;
use strict;
use parent qw(Point PlanarObject);

sub new { ... }

sub polar_coordinates { ... }

1;
```

La résolution de la méthode fonctionne comme suit:

1. Le point de départ est défini par l'opérande gauche de l'opérateur de flèche.

    - Si c'est un mot nu:

      ```
      Point2D->new(...);
      ```

      ... ou une variable scalaire contenant une chaîne:

      ```
      my $class = 'Point2D';
      $class->new(...);
      ```

      ... alors le point de départ est le package avec le nom correspondant (`Point2D` dans les deux exemples).

    - Si l'opérande gauche est une variable scalaire contenant une référence *bénie* :

      ```
      my $point = {...};
      bless $point, 'Point2D'; # typically, it is encapsulated into class methods
      my @coord = $point->polar_coordinates;
      ```

      alors le point de départ est la classe de la référence (encore une fois, `Point2D`). L'opérateur de flèche ne peut pas être utilisé pour appeler des méthodes pour les références non *bénies* .

2. Si le point de départ contient la méthode requise, il est simplement appelé.

    Ainsi, depuis `Point2D::new` existe,

    ```
    Point2D->new(...);
    ```

    l'appellera simplement

3. Si le point de départ ne contient pas la méthode requise, une recherche en profondeur dans les classes `parent` est effectuée. Dans l'exemple ci-dessus, l'ordre de recherche sera le suivant:

    - `Point2D`
    - `Point` (premier parent de `Point2D`)
    - `GeometryObject` (parent de `Point`)
    - `PlanarObject` (deuxième parent de `Point2D`)

    Par exemple, dans le code suivant:

    ```
    my $point = Point2D->new(...);
    $point->transpose(...);
    ```

    la méthode qui sera appelée est `GeometryObject::transpose` , même si elle serait remplacée

dans `PlanarObject::transpose` .

4. Vous pouvez définir le point de départ explicitement.

   Dans l'exemple précédent, vous pouvez appeler explicitement `PlanarObject::transpose` comme `PlanarObject::transpose` :

   ```
   my $point = Point2D->new(...);
   $point->PlanarObject::transpose(...);
   ```

5. De manière similaire, `SUPER::` effectue une recherche de méthode dans les classes parentes de la classe en cours.

   Par exemple,

   ```
   package Point2D;
   use strict;
   use parent qw(Point PlanarObject);

   sub new {
       (my $class, $x, $y) = @_;
       my $self = $class->SUPER::new;
       ...
   }

   1;
   ```

   appellera `Point::new` au cours de l'exécution de `Point2D::new` .

Méthodes de classe et d'objet

En Perl, la différence entre les méthodes de classe (statique) et d'objet (instance) n'est pas aussi forte que dans d'autres langages, mais elle existe toujours.

L'opérande gauche de l'opérateur de flèche `->` devient le premier argument de la méthode à appeler. Ce peut être soit une chaîne:

```
# the first argument of new is string 'Point' in both cases
Point->new(...);

my $class = 'Point';
$class->new(...);
```

ou une référence d'objet:

```
# reference contained in $point is the first argument of polar_coordinates
my $point = Point->new(...);
my @coord = $point->polar_coordinates;
```

Les méthodes de classe ne sont que celles qui attendent que leur premier argument soit une chaîne et que les méthodes d'objet soient celles qui s'attendent à ce que leur premier argument soit une référence d'objet.

Les méthodes de classe ne font généralement rien avec leur premier argument, qui n'est qu'un nom de classe. En général, Perl lui-même l'utilise uniquement pour la résolution de la méthode. Par conséquent, une méthode de classe typique peut également être appelée pour un objet:

```
my $width = Point->canvas_width;

my $point = Point->new(...);
my $width = $point->canvas_width;
```

Bien que cette syntaxe soit autorisée, elle est souvent trompeuse, il est donc préférable de l'éviter.

Les méthodes d'objet reçoivent une référence d'objet en tant que premier argument afin qu'elles puissent adresser les données d'objet (contrairement aux méthodes de classe):

```
package Point;
use strict;

sub polar_coordinates {
    my ($point) = @_;
    my $x = $point->{x};
    my $y = $point->{y};
    return (sqrt($x * $x + $y * $y), atan2($y, $x));
}

1;
```

La même méthode peut suivre les deux cas: lorsqu'elle est appelée en tant que classe ou méthode d'objet:

```
sub universal_method {
    my $self = shift;
    if (ref $self) {
        # object logic
        ...
    }
    else {
        # class logic
        ...
    }
}
```

Définir des classes en Perl moderne

Bien que disponible, la définition d'une classe à partir de zéro n'est pas recommandée dans Perl moderne. Utilisez l'un des systèmes OO d'assistance qui offrent plus de fonctionnalités et de commodité. Parmi ces systèmes sont:

- `Moose` - inspiré par le design Perl 6 OO
- `Class::Accessor` - une alternative légère à Moose
- `Class::Tiny` - constructeur de classes vraiment minimaliste

élan

```
package Foo;
use Moose;

has bar => (is => 'ro');                 # a read-only property
has baz => (is => 'rw', isa => 'Bool');  # a read-write boolean property

sub qux {
    my $self = shift;
    my $barIsBaz = $self->bar eq 'baz';  # property getter
    $self->baz($barIsBaz);               # property setter
}
```

Classe :: Accesseur (syntaxe Moose)

```
package Foo;
use Class::Accessor 'antlers';

has bar => (is => 'ro');                 # a read-only property
has baz => (is => 'rw', isa => 'Bool');  # a read-write property (only 'is' supported, the
type is ignored)
```

Class :: Accessor (syntaxe native)

```
package Foo;
use base qw(Class::Accessor);

Foo->mk_accessors(qw(bar baz));  # some read-write properties
Foo->mk_accessors(qw(qux));      # a read-only property
```

Classe :: Minuscule

```
package Foo;
use Class::Tiny qw(bar baz);  # just props
```

Les rôles

Un rôle dans Perl est essentiellement

- un ensemble de méthodes et d'attributs qui
- injecté directement dans une classe.

Un rôle fournit une fonctionnalité qui peut être *composée* (ou *appliquée* à) toute classe (qui est dite *consommer* le rôle). Un rôle ne peut pas être hérité mais peut être consommé par un autre rôle.

Un rôle peut également *nécessiter* des classes consommatrices pour implémenter certaines méthodes au lieu d'implémenter les méthodes elles-mêmes (tout comme les interfaces Java ou C #).

Perl ne prend pas en charge les rôles, mais il existe des classes CPAN qui fournissent un tel

support.

Moose :: Rôle

```
package Chatty;
use Moose::Role;

requires 'introduce';  # a method consuming classes must implement

sub greet {            # a method already implemented in the role
    print "Hi!\n";
}


package Parrot;
use Moose;

with 'Chatty';

sub introduce {
    print "I'm Buddy.\n";
}
```

Rôle :: Minuscule

Utilisez cette option si votre système OO ne prend pas en charge les rôles (par exemple, `Class::Accessor` ou `Class::Tiny`). Ne supporte pas les attributs.

```
package Chatty;
use Role::Tiny;

requires 'introduce';  # a method consuming classes must implement

sub greet {            # a method already implemented in the role
    print "Hi!\n";
}

package Parrot;
use Class::Tiny;
use Role::Tiny::With;

with 'Chatty';

sub introduce {
    print "I'm Buddy.\n";
}
```

Lire Perl orienté objet en ligne: https://riptutorial.com/fr/perl/topic/2920/perl-oriente-objet

Chapitre 31: Perlbrew

Introduction

Perlbrew est un outil pour gérer plusieurs installations de perl dans votre répertoire `$HOME` .

Remarques

Voir également

- Page d'accueil officielle pour perlbrew
- Documentation CPAN pour perlbrew

Examples

Configurez perlbrew pour la première fois

Créez le script d'installation `~/.perlbrew.sh` :

```
# Reset any environment variables that could confuse `perlbrew`:
export PERL_LOCAL_LIB_ROOT=
export PERL_MB_OPT=
export PERL_MM_OPT=

# decide where you want to install perlbrew:
export PERLBREW_ROOT=~/perlbrew
[[ -f "$PERLBREW_ROOT/etc/bashrc" ]] && source "$PERLBREW_ROOT/etc/bashrc"
```

Créez le script d'installation `install_perlbrew.sh` :

```
source ~/.perlbrew.sh
curl -L https://install.perlbrew.pl | bash
source "$PERLBREW_ROOT/etc/bashrc"

# Decide which version you would like to install:
version=perl-5.24.1
perlbrew install "$version"
perlbrew install-cpanm
perlbrew switch  "$version"
```

Exécuter le script d'installation:

```
./install_perlbrew.sh
```

Ajoutez à la fin de votre `~/.bashrc`

```
[[ -f ~/.perlbrew.sh ]] && source ~/.perlbrew.sh
```

Source `~/.bashrc` :

```
source ~/.bashrc
```

Lire Perlbrew en ligne: https://riptutorial.com/fr/perl/topic/9144/perlbrew

Chapitre 32: Séparer une chaîne sur des séparateurs non cotés

Examples

parse_line ()

Utilisation de `parse_line()` de Text :: ParseWords :

```
use 5.010;
use Text::ParseWords;

my $line = q{"a quoted, comma", word1, word2};
my @parsed = parse_line(',', 1, $line);
say for @parsed;
```

Sortie:

```
"a quoted, comma"
 word1
 word2
```

Text :: CSV ou Text :: CSV_XS

```
use Text::CSV; # Can use Text::CSV which will switch to _XS if installed
$sep_char = ",";
my $csv = Text::CSV->new({sep_char => $sep_char});
my $line = q{"a quoted, comma", word1, word2};
$csv->parse($line);
my @fields = $csv->fields();
print join("\n", @fields)."\n";
```

Sortie:

```
a quoted, comma
 word1
 word2
```

REMARQUES

- Par défaut, Text :: CSV ne `Text::ParseWords` pas les espaces autour du caractère séparateur, `Text::ParseWords` . Cependant, l'ajout de `allow_whitespace=>1` aux attributs du constructeur permet d'obtenir cet effet.

  ```
  my $csv = Text::CSV_XS->new({sep_char => $sep_char, allow_whitespace=>1});
  ```

Sortie:

```
a quoted, comma
word1
word2
```

- La bibliothèque prend en charge les caractères spéciaux échappés (guillemets, séparateurs)
- La bibliothèque prend en charge le caractère séparateur configurable, le caractère de citation et le caractère d'échappement

Documentatoin: http://search.cpan.org/perldoc/Text::CSV

Lire Séparer une chaîne sur des séparateurs non cotés en ligne: https://riptutorial.com/fr/perl/topic/2115/separer-une-chaine-sur-des-separateurs-non-cotes

Chapitre 33: Sous-routines

Remarques

Les sous-routines obtiennent leurs arguments dans la variable magique appelée `@_` . Bien qu'il ne soit pas nécessaire de le décompresser, il est recommandé, car il facilite la lecture et évite les modifications accidentelles, car les arguments de `@_` sont transmis par référence (peuvent être modifiés).

Examples

Créer des sous-routines

Les sous-routines sont créées en utilisant le mot-clé `sub` suivi d'un identifiant et d'un bloc de code entre accolades.

Vous pouvez accéder aux arguments en utilisant la variable spéciale `@_` , qui contient tous les arguments sous forme de tableau.

```
sub function_name {
    my ($arg1, $arg2, @more_args) = @_;
    # ...
}
```

Comme la fonction `shift` `@_` défaut le décalage de `@_` lorsqu'elle est utilisée dans un sous-programme, il est courant d'extraire les arguments séquentiellement dans des variables locales au début d'un sous-programme:

```
sub function_name {
    my $arg1 = shift;
    my $arg2 = shift;
    my @more_args = @_;
    # ...
}

# emulate named parameters (instead of positional)
sub function_name {
    my %args = (arg1 => 'default', @_);
    my $arg1 = delete $args{arg1};
    my $arg2 = delete $args{arg2};
    # ...
}

sub {
    my $arg1 = shift;
    # ...
}->($arg);
```

5.20.0

Alternativement, la caractéristique expérimentale `"signatures"` peut être utilisée pour décompresser les paramètres, qui sont passés par valeur (*pas* par référence).

```
use feature "signatures";

sub function_name($arg1, $arg2, @more_args) {
    # ...
}
```

Les valeurs par défaut peuvent être utilisées pour les paramètres.

```
use feature "signatures";

sub function_name($arg1=1, $arg2=2) {
    # ...
}
```

Vous pouvez utiliser n'importe quelle expression pour donner une valeur par défaut à un paramètre, y compris d'autres paramètres.

```
sub function_name($arg1=1, $arg2=$arg1+1) {
    # ...
}
```

Notez que vous ne pouvez pas référencer les paramètres définis après le paramètre actuel. Le code suivant ne fonctionne donc pas comme prévu.

```
sub function_name($arg1=$arg2, $arg2=1) {
    print $arg1;  # => <nothing>
    print $arg2;  # => 1
}
```

Les arguments de sous-routine sont transmis par référence (sauf ceux des signatures)

Les arguments de sous-routine en Perl sont transmis par référence, sauf s'ils se trouvent dans la signature. Cela signifie que les membres du tableau `@_` dans le sous-ensemble ne sont que des *alias* aux arguments réels. Dans l'exemple suivant, `$text` dans le programme principal reste modifié après l'appel du sous-programme car `$_[0]` à l'intérieur du sous-marin n'est en fait qu'un nom différent pour la même variable. La seconde invocation génère une erreur car un littéral de chaîne n'est pas une variable et ne peut donc pas être modifié.

```
use feature 'say';

sub edit {
    $_[0] =~ s/world/sub/;
}

my $text = "Hello, world!";
edit($text);
say $text;      # Hello, sub!
```

```
edit("Hello, world!"); # Error: Modification of a read-only value attempted
```

Pour éviter de frapper les variables de votre interlocuteur, il est donc important de copier `@_` vers des variables localisées (`my ...`) comme décrit dans "Création de sous-routines".

Sous-routines

Les sous-programmes contiennent du code. Sauf indication contraire, ils sont définis globalement.

```
# Functions do not (have to) specify their argument list
sub returns_one {
  # Functions return the value of the last expression by default
  # The return keyword here is unnecessary, but helps readability.
  return 1;
}

# Its arguments are available in @_, however
sub sum {
  my $ret = 0;
  for my $value (@_) {
    $ret += $value
  }
  return $ret;
}

# Perl makes an effort to make parens around argument list optional
say sum 1..3;     # 6

# If you treat functions as variables, the & sigil is mandatory.
say defined &sum; # 1
```

Certains builtins tels que l' `print` ou `say` sont des mots clés, pas les fonctions, donc par exemple `&say` est indéfini. Cela signifie également que vous pouvez les définir, mais vous devrez spécifier le nom du paquet pour les appeler réellement

```
# This defines the function under the default package, 'main'
sub say {
  # This is instead the say keyword
  say "I say, @_";
}

# ...so you can call it like this:
main::say('wow'); # I say, wow.
```

5.18.0

Depuis Perl 5.18, vous pouvez également avoir des fonctions non globales:

```
use feature 'lexical_subs';
my $value;
{
  # Nasty code ahead
  my sub prod {
    my $ret = 1;
```

```
    $ret *= $_ for @_;
    $ret;
  }
  $value = prod 1..6; # 720
  say defined &prod; # 1
}
say defined &prod; # 0
```

5.20.0

Depuis 5.20, vous pouvez également avoir des paramètres nommés.

```
use feature 'signatures';
sub greet($name) {
  say "Hello, $name";
}
```

Cela *ne* doit *pas* être confondu avec les prototypes, une installation que Perl doit vous permettre de définir des fonctions qui se comportent comme des éléments intégrés. Les prototypes de fonctions doivent être visibles au moment de la compilation et ses effets peuvent être ignorés en spécifiant le `&` sigil. Les prototypes sont généralement considérés comme une fonctionnalité avancée qui est mieux utilisée avec précaution.

```
# This prototype makes it a compilation error to call this function with anything
# that isn't an array. Additionally, arrays are automatically turned into arrayrefs
sub receives_arrayrefs(\@\@) {
   my $x = shift;
   my $y = shift;
}

my @a = (1..3);
my @b = (1..4);
receives_arrayrefs(@a, @b);    # okay,   $x = \@a, $y = \@b, @_ = ();
receives_arrayrefs(\@a, \@b);  # compilation error, "Type … must be array …"
BEGIN { receives_arrayrefs(\@a, \@b); }

# Specify the sigil to ignore the prototypes.
&receives_arrayrefs(\@a, \@b); # okay,   $x = \@a, $y = \@b, @_ = ();
&receives_arrayrefs(@a, @b);   # ok, but $x = 1,   $y = 2,   @_ = (3,1,2,3,4);
```

Lire Sous-routines en ligne: https://riptutorial.com/fr/perl/topic/711/sous-routines

Chapitre 34: Test Perl

Examples

Exemple de test d'unité Perl

Voici un exemple simple de script de test Perl, qui donne une structure permettant de tester d'autres méthodes dans la classe / le package testé. Le script produit une sortie standard avec un simple texte "ok" / "not ok", appelé TAP (Test Anything Protocol).

En général , la prouver commande exécute le script (s) et résume les résultats des tests.

```
#!/bin/env perl
# CPAN
use Modern::Perl;
use Carp;
use Test::More;
use Test::Exception;
use Const::Fast;

# Custom
BEGIN { use_ok('Local::MyPackage'); }

const my $PACKAGE_UNDER_TEST => 'Local::MyPackage';

# Example test of method 'file_type_build'
sub test_file_type_build {
    my %arg    = @_;
    my $label  = 'file_type_build';
    my $got_file_type;
    my $filename = '/etc/passwd';

    # Check the method call lives
    lives_ok(
        sub {
            $got_file_type = $PACKAGE_UNDER_TEST->file_type_build(
                filename => $filename
            );
        },
        "$label - lives"
    );

    # Check the result of the method call matches our expected result.
    like( $got_file_type, qr{ASCII[ ]text}ix, "$label - result" );
    return;
} ## end sub test_file_type_build

# More tests can be added here for method 'file_type_build', or other methods.


MAIN: {

   subtest 'file_type_build' => sub {
      test_file_type_build();
      # More tests of the method can be added here.
```

```
      done_testing();
   };

   # Tests of other methods can be added here, just like above.


   done_testing();
} ## end MAIN:
```

Meilleur entrainement

Un script de test ne doit tester qu'un seul package / classe, mais de nombreux scripts peuvent être utilisés pour tester un package / une classe.

Lectures complémentaires

- Test :: More - Les opérations de test de base.
- Test :: Exception - Test des exceptions levées.
- Test :: Differences - Comparaison des résultats de test avec des structures de données complexes.
- Test :: Class - Test basé sur la classe plutôt que script. Similitudes avec JUnit.
- Didacticiels de test Perl - Pour en savoir plus.

Lire Test Perl en ligne: https://riptutorial.com/fr/perl/topic/5918/test-perl

Chapitre 35: Texte Attribué

Examples

Impression de texte en couleur

```
#!/usr/bin/perl

use Term::ANSIColor;

print color("cyan"), "Hello", color("red"), "\tWorld", color("green"), "\tIt's Me!\n",
color("reset");
```

Lire Texte Attribué en ligne: https://riptutorial.com/fr/perl/topic/5922/texte-attribue

Chapitre 36: Tri

Introduction

Pour le tri des listes de choses, Perl n'a qu'une seule fonction, appelée sans surprise `sort` . Il est suffisamment flexible pour trier tous les types d'éléments: nombres, chaînes dans un nombre quelconque d'encodages, structures de données imbriquées ou objets. Cependant, en raison de sa flexibilité, il existe de nombreuses astuces et idiomes à utiliser pour son utilisation.

Syntaxe

- sort SUBNAME LIST
- sort BLOCK LIST
- trier la liste

Examples

Tri Lexique de base

```
@sorted = sort @list;

@sorted = sort { $a cmp $b } @list;

sub compare { $a cmp $b }
@sorted = sort compare @list;
```

Les trois exemples ci-dessus font exactement la même chose. Si vous ne fournissez aucune fonction ou bloc de comparaison, `sort` supposant que vous souhaitez que la liste à sa droite soit triée du point de vue lexical. C'est généralement la forme que vous voulez si vous avez simplement besoin de vos données dans un ordre prévisible et que vous ne vous souciez pas de la correction linguistique.

`sort` paires d'éléments dans `@list` à la fonction de comparaison, qui indique le `sort` élément le plus grand. L'opérateur `cmp` fait pour les chaînes alors que `<=>` fait la même chose pour les nombres. Le comparateur est appelé assez souvent, en moyenne *n* * log (*n*) fois, *n* étant le nombre d'éléments à trier, il est donc important qu'il soit rapide. C'est la raison pour laquelle le `sort` utilise des variables globales de package prédéfinies (`$a` et `$b`) pour passer les éléments à comparer au bloc ou à la fonction, au lieu de paramètres de fonction appropriés.

Si vous `use locale` , `cmp` prend en compte l'ordre de classement spécifique à l'environnement local, par exemple, il triera `Å` comme `A` sous une langue danoise mais après `Z` sous une langue anglaise ou allemande. Cependant, il ne prend pas en compte les règles de tri Unicode plus complexes et n'offre aucun contrôle sur la commande - par exemple, les annuaires téléphoniques sont souvent triés différemment des dictionnaires. Pour ces cas, les modules `Unicode::Collate` et particulièrement `Unicode::Collate::Locale` sont recommandés.

Numérique Sort

```
@sorted = sort { $a <=> $b } @list;
```

En comparant `$a` et `$b` à l'opérateur `<=>` , on s'assure qu'ils sont comparés numériquement et non textuellement selon la valeur par défaut.

Tri inverse

```
@sorted = sort { $b <=> $a } @list;
@sorted = reverse sort { $a <=> $b } @list;
```

Le tri des éléments en ordre décroissant peut être réalisé simplement en échangeant `$a` et `$b` dans le bloc comparateur. Cependant, certaines personnes préfèrent la clarté d'un `reverse` séparé, même s'il est légèrement plus lent.

La Transformation Schwartzienne

C'est probablement l'exemple le plus célèbre d'une optimisation de tri utilisant les fonctions de programmation fonctionnelle de Perl, à utiliser lorsque l'ordre de tri des éléments dépend d'une fonction coûteuse.

```
# What you would usually do
@sorted = sort { slow($a) <=> slow($b) } @list;

# What you do to make it faster
@sorted =
map { $_->[0] }
sort { $a->[1] <=> $b->[1] }
map { [ $_, slow($_) ] }
@list;
```

Le problème avec le premier exemple est que le comparateur est appelé très souvent et continue à recalculer les valeurs en utilisant une fonction lente encore et encore. Un exemple typique serait le tri des noms de fichiers par la taille de leur fichier:

```
use File::stat;
@sorted = sort { stat($a)->size <=> stat($b)->size } glob "*";
```

Cela fonctionne, mais au mieux, il subit la surcharge de deux appels système par comparaison, au pire il doit aller au disque, deux fois, pour chaque comparaison, et ce disque peut se trouver dans un serveur de fichiers surchargé de l'autre côté du disque. planète.

Entrez le tour de Randall Schwartz.

La transformation Schwartzian transforme fondamentalement `@list` en trois fonctions, de bas en haut. La première `map` transforme chaque entrée en une liste de deux éléments de l'élément d'origine et le résultat de la fonction lente en tant que clé de tri. Nous avons donc appelé `slow()` une fois pour chaque élément. Le `sort` suivant peut alors simplement accéder à la clé de tri en

consultant la liste. Comme nous ne nous soucions pas des clés de tri mais que nous avons seulement besoin des éléments d'origine dans l'ordre trié, la `map` finale jette les listes à deux éléments de la liste déjà triée `@sort` de `@sort` et renvoie uniquement la liste de leurs premiers membres .

Tri insensible à la casse

La technique traditionnelle pour faire `sort` cas est de passer des chaînes à `lc` ou `uc` pour comparaison:

```
@sorted = sort { lc($a) cmp lc($b) } @list;
```

Cela fonctionne sur toutes les versions de Perl 5 et est tout à fait suffisant pour l'anglais; Peu importe si vous utilisez `uc` ou `lc` . Cependant, cela pose un problème pour les langues comme le grec ou le turc où il n'y a pas de correspondance 1: 1 entre les lettres majuscules et minuscules, ce qui vous donne des résultats différents selon que vous utilisez `uc` ou `lc` . Par conséquent, Perl 5.16 et les versions ultérieures ont une fonction de *pliage de cas* appelée `fc` qui évite ce problème, de sorte que le tri multilingue moderne devrait utiliser ceci:

```
@sorted = sort { fc($a) cmp fc($b) } @list;
```

Lire Tri en ligne: https://riptutorial.com/fr/perl/topic/8958/tri

Chapitre 37: Un moyen facile de vérifier les modules installés sur Mac et Ubuntu

Examples

Vérifiez les modules Perl installés via le terminal

Tapez la commande ci-dessous:

```
instmodsh
```

Il vous montrera la guilde ci-dessous:

```
Available commands are:
   l            - List all installed modules
   m <module>   - Select a module
   q            - Quit the program
cmd?
```

Puis tapez `l` pour lister tous les modules installés, vous pouvez également utiliser la commande `m <module>` pour sélectionner le module et obtenir ses informations.

Après avoir fini, tapez simplement `q` pour quitter.

Utilisez perldoc pour vérifier le chemin d'installation du paquet Perl

```
$ perldoc -l Time::Local
```

Comment vérifier les modules de corelist Perl.

```
$ corelist -v v5.23.1
```

Comment vérifier la version d'un module installé?

```
$> perl -MFoo::Bar\ 9999
$> Foo::Bar version 9999 required--this is only version 1.1.
```

Lire Un moyen facile de vérifier les modules installés sur Mac et Ubuntu en ligne: https://riptutorial.com/fr/perl/topic/5925/un-moyen-facile-de-verifier-les-modules-installes-sur-mac-et-ubuntu

Chapitre 38: Unicode

Remarques

Un avertissement sur le codage des noms de fichiers

Il convient de mentionner que le codage de nom de fichier n'est pas seulement spécifique à la *plate-forme,* mais aussi au *système de fichiers* .

Il n'est jamais *tout à fait* sûr de supposer (mais c'est souvent le cas) que le simple fait que vous puissiez encoder et écrire sur un nom de fichier donné, lorsque vous essayez d'ouvrir le même nom pour la lecture, sera toujours appelé.

Par exemple, si vous écrivez dans un système de fichiers tel que `FAT16` qui ne prend pas en charge unicode, vos noms de fichiers peuvent être traduits en silence dans des formulaires compatibles ASCII.

Mais il est encore *moins* sûr de supposer qu'un fichier , vous pouvez créer, lire et écrire en nommant explicitement que l' on appellera la même chose lorsqu'il est interrogé par d' autres appels, par exemple, `readdir` peut renvoyer différents octets pour votre nom de fichier que vous avez spécifié pour `open` .

Sur certains systèmes tels que VAX, vous ne pouvez pas toujours supposer que `readdir` renverra le même nom de fichier que vous avez spécifié avec `open` pour les noms de fichiers aussi simples que `foo.bar` , car les *extensions de* noms de fichiers peuvent être gérées par le système d'exploitation.

De plus, sous UNIX, il existe un ensemble très libéral de caractères légaux pour les noms de fichiers que le système d'exploitation autorise, à l'exception de `/` et `\0` , comme sur Windows, il existe des plages de caractères interdites dans les noms de fichiers.

Faites preuve de **prudence beaucoup ici,** d' **éviter** des **trucs** de **fantaisie avec** les **noms** de **fichiers si vous avez le choix,** et ont toujours des tests pour vous assurer que tous les trucs de fantaisie que vous *faites* usage sont compatibles.

Faites preuve d'autant de prudence si vous écrivez du code destiné à être exécuté sur des plates-formes hors de votre contrôle, par exemple si vous écrivez du code destiné à `CPAN` , et supposez qu'au moins 5% de votre base d'utilisateurs sera bloquée par certains. une technologie ancienne ou brisée, que ce soit par choix, par accident ou par des pouvoirs indépendants de leur volonté, et que ceux-ci concourront à créer des bogues pour eux.

: encodage (utf8) vs: utf8

Comme UTF-8 est l'un des formats internes de représentation des chaînes en Perl, l'étape de codage / décodage peut souvent être ignorée. Au lieu de `:encoding(utf-8)` , vous pouvez simplement utiliser `:utf8` , si vos données sont déjà dans UTF-8. `:utf8` peut être utilisé en toute sécurité avec les flux de sortie, alors que pour le flux d'entrée, il peut être dangereux, car il provoque une incohérence interne lorsque vous avez des séquences d'octets invalides. En outre, l'utilisation de `:utf8` pour la saisie peut entraîner des failles de sécurité. L'utilisation de `:encoding(utf-8)` est donc recommandée.

Plus de détails: Quelle est la différence entre: encoding et: utf8

UTF-8 vs utf8 vs UTF8

A partir de Perl `v5.8.7` , `"UTF-8"` (avec un tiret) signifie UTF-8 dans sa forme stricte et consciente de la sécurité, tandis que `"utf8"` signifie UTF-8 dans sa forme libérale et libre.

Par exemple, `"utf8"` peut être utilisé pour les points de code qui n'existent pas dans Unicode, comme `0xFFFFFFFF` . De manière correspondante, les séquences d'octets UTF-8 invalides comme `"\x{FE}\x{83}\x{BF}\x{BF}\x{BF}\x{BF}\x{BF}"` se décoderont en un invalide codepoint Unicode (mais Perl valide) (`0xFFFFFFFF`) lors de l' utilisation `"utf8"` , alors que le `"UTF-8"` le codage ne permettrait pas le décodage de codepoints en dehors de la plage d'Unicode valide et vous donnera un caractère de substitution (`0xFFFD`) à la place.

Comme les noms de codage sont insensibles à la casse, `"UTF8"` est identique à `"utf8"` (variante *non stricte*).

Plus de détails: UTF-8 contre utf8 contre UTF8

Plus de lecture

Les détails sur la gestion Unicode de Perl sont décrits plus en détail dans les sources suivantes:

- perlunicode
- perlunitut
- perluniintro
- perlunifaq
- perlunicook
- utf8 pragma
- fonctionnalité unicode_strings
- pragma ouvert
- PerlIO

- PerlIO :: encodage
- fonction ouverte
- Encoder
- perlrun - commutateurs de ligne de commande
- Chapitre 6, Programmation Perl

Messages de stackoverflow.com (mise en garde: peut ne pas être à jour):

- Pourquoi Perl moderne évite-t-il UTF-8 par défaut?

Vidéos youtube:

- Un million de milliards de personnages sans soucis par Ricardo Signes à YAPC NA 2016.

Examples

Créer des noms de fichiers

Les exemples suivants utilisent le codage UTF-8 pour représenter les noms de fichiers (et les noms de répertoires) sur le disque. Si vous voulez utiliser un autre encodage, vous devez utiliser `Encode::encode(...)` .

```
use v5.14;
# Make Perl recognize UTF-8 encoded characters in literal strings.
# For this to work: Make sure your text-editor is using UTF-8, so
# that bytes on disk are really UTF-8 encoded.
use utf8;

# Ensure that possible error messages printed to screen are converted to UTF-8.
# For this to work: Check that your terminal emulator is using UTF-8.
binmode STDOUT, ':utf8';
binmode STDERR, ':utf8';

my $filename = 'æ€'; # $filename is now an internally UTF-8 encoded string.

# Note: in the following it is assumed that $filename has the internal UTF-8
#  flag set, if $filename is pure ASCII, it will also work since its encoding
#  overlaps with UTF-8. However, if it has another encoding like extended ASCII,
#  $filename will be written with that encoding and not UTF-8.
# Note: it is not necessary to encode $filename as UTF-8 here
#  since Perl is using UTF-8 as its internal encoding of $filename already

# Example1  -- using open()
open ( my $fh, '>', $filename ) or die "Could not open '$filename': $!";
close $fh;

# Example2 -- using qx() and touch
qx{touch $filename};

# Example3 -- using system() and touch
system 'touch', $filename;

# Example4 -- using File::Touch
use File::Touch;
eval { touch( $filename ) }; die "Could not create file '$filename': $!" if $@;
```

Lire les noms de fichiers

Perl ne tente pas de décoder les noms de fichiers renvoyés par des fonctions ou des modules intégrés. De telles chaînes représentant des noms de fichiers doivent toujours être décodées explicitement, afin que Perl les reconnaisse comme Unicode.

```
use v5.14;
use Encode qw(decode_utf8);

# Ensure that possible error messages printed to screen are converted to UTF-8.
# For this to work: Check that you terminal emulator is using UTF-8.
binmode STDOUT, ':utf8';
binmode STDERR, ':utf8';

# Example1  -- using readdir()
my $dir = '.';
opendir(my $dh, $dir) or die "Could not open directory '$dir': $!";
while (my $filename = decode_utf8(readdir $dh)) {
    # Do something with $filename
}
close $dh;

# Example2  -- using getcwd()
use Cwd qw(getcwd);
my $dir = decode_utf8( getcwd() );

# Example3  -- using abs2rel()
use File::Spec;
use utf8;
my $base = 'ø';
my $path = "$base/b/æ";
my $relpath = decode_utf8( File::Spec->abs2rel( $path, $base ) );
# Note: If you omit $base, you need to encode $path first:
use Encode qw(encode_utf8);
my $relpath = decode_utf8( File::Spec->abs2rel( encode_utf8( $path ) ) );

# Example4  -- using File::Find::Rule (part1 matching a filename)
use File::Find::Rule;
use utf8;
use Encode qw(encode_utf8);
my $filename = 'æ';
# File::Find::Rule needs $filename to be encoded
my @files = File::Find::Rule->new->name( encode_utf8($filename) )->in('.');
$_ = decode_utf8( $_ ) for @files;

# Example5  -- using File::Find::Rule (part2 matching a regular expression)
use File::Find::Rule;
use utf8;
my $pat = '[æ].$'; # Unicode pattern
# Note: In this case:  File::Find::Rule->new->name( qr/$pat/ )->in('.')
#  will not work since $pat is Unicode and filenames are bytes
#  Also encoding $pat first will not work correctly
my @files;
File::Find::Rule->new->exec( sub { wanted( $pat, \@files ) } )->in('.');
$_ = decode_utf8( $_ ) for @files;
sub wanted {
    my ( $pat, $files ) = @_;
    my $name = decode_utf8( $_ );
    my $full_name = decode_utf8( $File::Find::name );
```

```
    push @$files, $full_name if $name =~ /$pat/;
}
```

Note: si vous êtes préoccupé par l'invalidité de l'UTF-8 dans les noms de fichiers, l'utilisation de `decode_utf8( ... )` dans les exemples ci-dessus devrait probablement être remplacée par `decode( 'utf-8', ... )` . En effet, `decode_utf8( ... )` est synonyme de `decode( 'utf8', ... )` et il existe une différence entre les encodages `utf-8` et `utf8` (voir Remarques ci-dessous pour plus d'informations) où `utf-8` est plus strict sur ce qui est acceptable que `utf8` .

Commutateurs de ligne de commande pour un interlocuteur

Activer le pragma utf8

Pour activer le pragma `utf8` dans une ligne, l'interpréteur perl doit être appelé avec l'option `-Mutf8` :

```
perl -Mutf8 -E 'my $□ = "human"; say $□'
```

Manipulation Unicode avec le commutateur -C

L'indicateur de ligne de commande `-C` vous permet de contrôler les fonctionnalités Unicode. Il peut être suivi d'une liste de lettres d'options.

E / S standard

- `I` - `STDIN` sera en *UTF-8*
- `O` - `STDOUT` sera en *UTF-8*
- `E` - `STDERR` sera en *UTF-8*
- `S` - raccourci pour `IOE` , les flux d'E / S standard seront en *UTF-8*

```
echo "Ματαιότης ματαιοτήτων" | perl -CS -Mutf8 -nE 'say "ok" if /Ματαιότης/'
```

Les arguments du script

- `A` - traite `@ARGV` comme un tableau de chaînes codées en *UTF-8*

```
perl -CA -Mutf8 -E 'my $arg = shift; say "anteater" if $arg eq "муравьед"' муравьед
```

Couche PerlIO par défaut

- `i` - *UTF-8* est la couche PerlIO par défaut pour les flux d'entrée
- `o`

- *UTF-8* est la couche PerlIO par défaut pour les flux de sortie

- `D` - sténographie pour `io`

```
perl -CD -Mutf8 -e 'open my $fh, ">", "utf8.txt" or die $!; print $fh "□□ □□□"'
```

`-M` commutateurs `-M` et `-C` peuvent être combinés:

```
perl -CASD -Mutf8 -E 'say "Ματαιότης ματαιοτήτων\n"';
```

E / S standard

Le codage à utiliser pour les `STDIN` E / S standard (`STDIN` , `STDOUT` et `STDERR`) peut être défini séparément pour chaque `binmode` aide de `binmode` :

```
binmode STDIN, ':encoding(utf-8)';
binmode STDOUT, ':utf8';
binmode STDERR, ':utf8';
```

Note: en lecture on préférerait en général `:encoding(utf-8)` sur `:utf8` , voir Remarques pour plus d'informations.

Alternativement, vous pouvez utiliser le pragma `open` .

```
# Setup such that all subsequently opened input streams will use ':encoding(utf-8)'
# and all subsequently opened output streams will use ':utf8'
# by default
use open (IN => ':encoding(utf-8)', OUT => ':utf8');
# Make the (already opened) standard file handles inherit the setting
# given by the IO settings for the open pragma
use open ( :std );
# Now, STDIN has been converted to ':encoding(utf-8)', and
# STDOUT and STDERR have ':utf8'
```

Alternativement, pour définir tous les descripteurs de fichiers (à la fois ceux qui doivent encore être ouverts et ceux à ouvrir) `:encoding(utf-8)` :

```
use open qw( :encoding(utf-8) :std );
```

Poignées de fichier

Réglage de l'encodage avec open ()

Lors de l'ouverture d'un fichier texte, vous pouvez spécifier son encodage explicitement avec un `open()` trois arguments. Ce en / décodeur attaché à un descripteur de fichier est appelé "couche d'E / S":

```
my $filename = '/path/to/file';
```

```
open my $fh, '<:encoding(utf-8)', $filename or die "Failed to open $filename: $!";
```

Voir Remarques pour une discussion des différences entre `:utf8` et `:encoding(utf-8)` .

Définition du codage avec binmode ()

Vous pouvez également utiliser binmode () pour définir l'encodage pour chaque descripteur de fichier:

```
my $filename = '/path/to/file';
open my $fh, '<', $filename or die "Failed to open $filename: $!";
binmode $fh, ':encoding(utf-8)';
```

pragma ouvert

Pour éviter de définir séparément le codage pour chaque descripteur de fichier, vous pouvez utiliser le pragma `open` pour définir une couche d'E / S par défaut utilisée par tous les appels suivants à la fonction `open()` et aux opérateurs similaires dans la portée lexicale de ce pragma:

```
# Set input streams to ':encoding(utf-8)' and output streams to ':utf8'
use open (IN => ':encoding(utf-8)', OUT => ':utf8');
# Or to set all input and output streams to ':encoding(utf-8)'
use open ':encoding(utf-8)';
```

Définition du codage avec la ligne de commande -C flag

Enfin, il est également possible d'exécuter l'interpréteur perl avec un indicateur `-CD` qui applique UTF-8 comme couche d'E / S par défaut. Cependant, cette option doit être évitée car elle repose sur un comportement spécifique de l'utilisateur qui ne peut être ni prédit ni contrôlé.

Le pragma utf8: utiliser Unicode dans vos sources

Le pragma `utf8` indique que le code source sera interprété comme UTF-8. Bien sûr, cela ne fonctionnera que si votre éditeur de texte enregistre également la source au format UTF-8.

Maintenant, les littéraux de chaîne peuvent contenir des caractères Unicode arbitraires. les identificateurs peuvent également contenir Unicode, mais uniquement des caractères de type mot (voir perldata et perlrecharclass pour plus d'informations):

```
use utf8;
my $var1 = '$я$©';      # works fine
my $я = 4;               # works since я is a word (matches \w) character
my $p§2 = 3;             # does not work since § is not a word character.
```

```
say "ya" if $var1 =~ /я§/; # works fine (prints "ya")
```

Remarque : Lorsque vous imprimez du texte sur le terminal, assurez-vous qu'il prend en charge UTF-8. *

Il peut exister des relations complexes et contre-intuitives entre la sortie et le codage source. En cours d'exécution sur un terminal UTF-8, vous pouvez constater que l'ajout du pragma `utf8` semble casser des choses:

```
$ perl -e 'print "Møøse\n"'
Møøse
$ perl -Mutf8 -e 'print "Møøse\n"'
M◆◆se
$ perl -Mutf8 -CO -e 'print "Møøse\n"'
Møøse
```

Dans le premier cas, Perl traite la chaîne comme des octets bruts et les imprime comme cela. Comme ces octets sont valides UTF-8, ils semblent corrects même si Perl ne sait pas vraiment quels caractères ils sont (par exemple, la `length("Møøse")` renverra 7, pas 5). Une fois que vous ajoutez `-Mutf8` , Perl décode correctement la source UTF-8 en caractères, mais la sortie est en mode Latin-1 par défaut et l'impression Latin-1 sur un terminal UTF-8 ne fonctionne pas. Ce n'est que lorsque vous passez de `STDOUT` à UTF-8 en utilisant `-CO` que la sortie sera correcte.

`use utf8` n'affecte pas l'encodage d'E / S standard ni les descripteurs de fichiers!

Gestion des UTF-8 invalides

Lecture invalide UTF-8

Lors de la lecture de données encodées en UTF-8, il est important de savoir que les données encodées en UTF-8 peuvent être invalides ou mal formées. Ces données ne devraient généralement pas être acceptées par votre programme (sauf si vous savez ce que vous faites). En cas de rencontre inopinée de données mal formées, différentes actions peuvent être envisagées:

- Imprimer le stacktrace ou le message d'erreur, et abandonner le programme normalement, ou
- Insérez un caractère de substitution à l'endroit où la séquence d'octets mal formée est apparue, imprimez un message d'avertissement à STDERR et continuez à lire car rien ne s'est produit.

Par défaut, Perl vous `warn` de l'encodage des erreurs, mais il n'abandonnera pas votre programme. Vous pouvez faire votre programme abort en faisant des avertissements UTF-8 fatale, mais être au courant des mises en garde Avertissements Fatal .

L'exemple suivant écrit 3 octets dans le codage ISO 8859-1 sur le disque. Il essaie ensuite de relire les octets en tant que données encodées en UTF-8. L'un des octets, `0xE5` , est une séquence d'octets UTF-8 invalide:

```
use strict;
use warnings;
use warnings FATAL => 'utf8';

binmode STDOUT, ':utf8';
binmode STDERR, ':utf8';
my $bytes = "\x{61}\x{E5}\x{61}";  # 3 bytes in iso 8859-1: aåa
my $fn = 'test.txt';
open ( my $fh, '>:raw', $fn ) or die "Could not open file '$fn': $!";
print $fh $bytes;
close $fh;
open ( $fh, "<:encoding(utf-8)", $fn ) or die "Could not open file '$fn': $!";
my $str = do { local $/; <$fh> };
close $fh;
print "Read string: '$str'\n";
```

Le programme sera interrompu par un avertissement fatal:

```
utf8 "\xE5" does not map to Unicode at ./test.pl line 10.
```

La ligne 10 est la deuxième dernière ligne, et l'erreur se produit dans la partie de la ligne avec `<$fh>` lorsque vous essayez de lire une ligne du fichier.

Si vous ne faites pas d'avertissement fatal dans le programme ci-dessus, Perl imprimera toujours l'avertissement. Cependant, dans ce cas, il essaiera de récupérer l'octet mal formé `0xE5` en insérant les quatre caractères `\xE5` dans le flux, puis poursuivra l'octet suivant. En conséquence, le programme imprimera:

```
Read string: 'a\xE5a'
```

Lire Unicode en ligne: https://riptutorial.com/fr/perl/topic/4375/unicode

Chapitre 39: Variables spéciales

Remarques

À FAIRE: Ajoutez plus de contenu.

Examples

Variables spéciales en perl:

1. `$_` : L'entrée par défaut et l'espace de recherche de modèle.

Exemple 1:

```
my @array_variable = (1 2 3 4);
foreach (@array_variable){
    print $_."\n";    # $_ will get the value 1,2,3,4 in loop, if no other variable is
supplied.
}
```

Exemple 2:

```
while (<FH>){
    chomp($_);    # $_ refers to the iterating lines in the loop.
}
```

Les fonctions suivantes utilisent `$_` comme argument par défaut:

```
abs, alarm, chomp, chop, chr, chroot, cos, defined, eval,
evalbytes, exp, fc, glob, hex, int, lc, lcfirst, length, log,
lstat, mkdir, oct, ord, pos, print, printf, quotemeta, readlink,
readpipe, ref, require, reverse (in scalar context only), rmdir,
say, sin, split (for its second argument), sqrt, stat, study,
uc, ucfirst, unlink, unpack.
```

2. `@_` : Ce tableau contient les arguments passés à la sous-routine.

Exemple 1:

```
example_sub( $test1, $test2, $test3 );

sub example_sub {
    my ( $test1, $test2, $test3 ) = @_;
}
```

Dans un sous-programme, le tableau `@_` contient les **arguments** transmis à ce sous-programme. Dans une sous-routine, `@_` est le tableau `default` pour les opérateurs de `pop` et les `shift` .

Lire Variables spéciales en ligne: https://riptutorial.com/fr/perl/topic/7962/variables-speciales

Chapitre 40: Vrai et faux

Syntaxe

- undef # faux
- '' # Défini, faux
- 0 # défini, a la longueur, faux
- '0' # défini, a la longueur, faux

Remarques

Perl n'a pas de type de données booléen, et n'a pas non plus de mots-clés `true` et `false` comme beaucoup d'autres langues. Cependant, chaque valeur scalaire est évaluée à true ou false lorsqu'elle est évaluée dans un contexte booléen (la condition dans une instruction if ou une boucle while, par exemple).

Les valeurs suivantes sont considérées comme fausses:

- `''` , la chaîne vide. C'est ce que les opérateurs de comparaison intégrés renvoient (par exemple, `0 == 1`)
- `0` , le nombre 0, même si vous écrivez comme 000 ou 0.0
- `'0'` , la chaîne qui contient un seul 0 chiffre
- `undef` , la valeur indéfinie
- Objets qui utilisent la surcharge pour numify / stringify en valeurs fausses, telles que `JSON::false`

Toutes les autres valeurs sont vraies:

- tout nombre non nul tel que `1` , `3.14` , `'NaN'` ou `'Inf'`
- n'importe quelle chaîne qui est numériquement 0 mais pas littéralement la chaîne `'0'` , telle que `'00'` , `'0e0'` , `"0\n"` et `"abc"` .
  Si vous retournez *intentionnellement* une vraie valeur numérique, préférez `'0E0'` (utilisé par les modules bien connus) ou `'0 but true'` (utilisé par les fonctions Perl)
- toute autre chaîne qui n'est pas vide, telle que `' '` , `'false'`
- toutes les références, même si elles font référence à des valeurs fausses, telles que `\''` , `[]` ou `{}`
- un tableau ou un hachage de valeurs fausses

Les opérateurs suivants sont généralement traités pour renvoyer un booléen dans un contexte scalaire:

- `@a` si le tableau est vide ou non

- `%h` retourne si le hachage est vide ou non
- `grep` retourne si des éléments correspondants ont été trouvés ou non
- `@a = LIST` et `(LIST) = LIST` renvoie si la liste de droite a produit des scalaires ou non

Examples

Liste des valeurs vraies et fausses

```
use feature qw( say );

# Numbers are true if they're not equal to 0.
say 0             ? 'true' : 'false'; # false
say 1             ? 'true' : 'false'; # true
say 2             ? 'true' : 'false'; # true
say -1            ? 'true' : 'false'; # true
say 1-1           ? 'true' : 'false'; # false
say 0e7           ? 'true' : 'false'; # false
say -0.00         ? 'true' : 'false'; # false

# Strings are true if they're not empty.
say 'a'           ? 'true' : 'false'; # true
say 'false'       ? 'true' : 'false'; # true
say ''            ? 'true' : 'false'; # false

# Even if a string would be treated as 0 in numeric context, it's true if nonempty.
# The only exception is the string "0", which is false.
# To force numeric context add 0 to the string
say '0'           ? 'true' : 'false'; # false
say '0.0'         ? 'true' : 'false'; # true
say '0e0'         ? 'true' : 'false'; # true
say '0 but true'  ? 'true' : 'false'; # true
say '0 whargarbl' ? 'true' : 'false'; # true
say 0+'0 argarbl' ? 'true' : 'false'; # false

# Things that become numbers in scalar context are treated as numbers.
my @c = ();
my @d = (0);
say @c            ? 'true' : 'false'; # false
say @d            ? 'true' : 'false'; # true

# Anything undefined is false.
say undef         ? 'true' : 'false'; # false

# References are always true, even if they point at something false
my @c = ();
my $d = 0;
say \@c            ? 'true' : 'false'; # true
say \$d            ? 'true' : 'false'; # true
say \0             ? 'true' : 'false'; # true
say \''            ? 'true' : 'false'; # true
```

Lire Vrai et faux en ligne: https://riptutorial.com/fr/perl/topic/649/vrai-et-faux

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec le langage Perl	Alan Haggai Alavi, choroba, Christopher Bottoms, Community, datageist, Denis Ibaev, eddy85br, Eugen Konkov, Jon Ericson, Leon Timmermans, oals, Pro Q, rlandster, xfix
2	Analyse XML	cbmckay, Drav Sloan, eballes, Sobrique
3	Applications GUI en Perl	oldtechaa
4	Au hasard	Christopher Bottoms, Rebecca Close, Zaid
5	Chaînes et méthodes de citation	badp, Christopher Bottoms, Denis Ibaev, digitalis_, Kent Fredric, mbethke, svarog
6	Commandes Perl pour Windows Excel avec le module Win32 :: OLE	Jean-Francois T.
7	commentaires	4444, Christopher Bottoms, lanti, Rebecca Close
8	Compiler le module cpan Perl sapnwrfc à partir du code source	flotux
9	Danseur	Chankey Pathak, vanHoesel
10	Dates et heure	Ngoan Tran, waghso
11	Débogage du script Perl	4444, Eugen Konkov
12	Déboguer la sortie	Ataul Haque, Christopher Bottoms, Joe, simbabque, waghso
13	Des listes	brian d foy, Christopher Bottoms, David Mertens, Denis Ibaev, DVK, Eugen Konkov, Muaaz Rafi, pwes, reflective_mind, Rick James, Wolf
14	Emballer et déballer	Denis Ibaev, Kent Fredric, mbethke
15	Expressions régulières	Al.G., Jon Ericson, rlandster, SajithP, Sarwesh Suman, Stephen Leppik

16	Fichier I / O (lecture et écriture de fichiers)	Christopher Bottoms, Denis Ibaev, Håkon Hægland, Kemi, Kent Fredric, matt freake, Nagaraju, rbennett485, SajithP, Sebi, SREagle, Tim Hallyburton, yonyon100
17	Gestion des exceptions	badp, simbabque
18	Installation de Perl	fanlim, flamey, Håkon Hægland, Iván Rodríguez Torres, luistm
19	Installer les modules Perl via CPAN	Christopher Bottoms, Kemi, luistm, Ngoan Tran, Peter Mortensen, Randall
20	Instructions de contrôle	callyalater, Christopher Bottoms, oals, Stephen Leppik
21	Interaction simple avec la base de données via le module DBI	Ngoan Tran
22	Interpolation en Perl	oals, Ruslan Batdalov
23	Lecture du contenu d'un fichier dans une variable	Alien Life Form, Christopher Bottoms, digitalis_, Jeff Y, Kemi, mbethke, mob, pwes, rlandster, SREagle
24	Les meilleures pratiques	fifaltra, interduo
25	Les variables	Ataul Haque, badp, digitalis_, dmvrtx, Eugen Konkov, Håkon Hægland, interduo, Jon Ericson, Kent Fredric, mbethke, Mik, nfanta, oals, Otterbein, zb226
26	Optimisation de l'utilisation de la mémoire	mbethke
27	Packages et modules	AntonH, Christopher Bottoms, John Hart, Jon Ericson, Kemi, Kent Fredric, lepe, mbethke
28	Perl one-liners	Dmitry Egorov, Eugen Konkov, Kemi, mbethke, zb226
29	Perl orienté objet	badp, Dmitry Egorov, Ruslan Batdalov, simbabque
30	Perlbrew	Håkon Hægland
31	Séparer une chaîne sur des séparateurs non cotés	DVK, Ian Praxil, serenesat

32	Sous-routines	badp, Christopher Bottoms, dave, digitalis_, interduo, mbethke, Michael Carman, msh210, Wolf, xfix, xtreak
33	Test Perl	nslntmnx
34	Texte Attribué	SajithP
35	Tri	Jon Ericson, kjpires, mbethke
36	Un moyen facile de vérifier les modules installés sur Mac et Ubuntu	fanlim, Ngoan Tran,
37	Unicode	Håkon Hægland, Kemi, Kent Fredric, mbethke
38	Variables spéciales	AbhiNickz, Denis Ibaev, oals
39	Vrai et faux	badp, Bill the Lizard, Christopher Bottoms, ikegami, Kent Fredric , mbethke, msh210, Ole Tange, xfix