# Perl Snelcursus:

---

Hier is de fundamentele perl programma dat we zullen gebruiken om te beginnen.

```
 #!/usr/local/bin/perl # # Program to do the obvious # print 'Hello world.';  # Print a message #! / Usr / local / bin / perl # # Program te doen voor de hand liggende # print "Hello world."; # Print een bericht
```

Each of the parts will be discussed in turn. Elk van de onderdelen zal worden besproken in de bocht.

---

## The first line De eerste regel

Every perl program starts off with this as its very first line: Elke perl programma begint met dit als haar allereerste regel:

```
 #!/usr/local/bin/perl #! / Usr / local / bin / perl
```

although this may vary from system to system. Maar dit kan verschillen van systeem tot systeem. This line tells the machine what to do with the file when it is executed (ie it tells it to run the file through Perl). Deze regel vertelt de machine wat te doen met het bestand wanneer het wordt uitgevoerd (dwz het geeft hem aan het bestand op te starten via Perl).

---

## Comments and statements Reacties en verklaringen

Comments can be inserted into a program with the # symbol, and anything from the # to the end of the line is ignored (with the exception of the first line). Reacties kunnen worden ingevoegd in een programma met de # symbool, en alles wat uit de # aan het einde van de regel wordt genegeerd (met uitzondering van de eerste lijn). The only way to stretch comments over several lines is to use a # on each line. De enige manier om stretch opmerkingen over meerdere lijnen is het gebruik van een # op elke regel.

Everything else is a Perl statement which must end with a semicolon, like the last line above. Al het andere is een Perl verklaring die moet eindigen met een puntkomma, zoals de laatste regel boven.

---

## Simple printing Eenvoudig printen

The `print` function outputs some information. De `print` functie uitgangen sommige informatie. In the above case it prints out the the literal string *Hello world.* and of course the statement ends

with a semicolon. In het bovenstaande geval geeft de letterlijke string *Hallo wereld.* En natuurlijk de verklaring eindigt met een puntkomma.

You may find the above program produces an slightly unexpected result. Je vindt het bovenstaande programma geeft een enigszins onverwacht resultaat. So the next thing to do is to run it. Dus het volgende wat je moet doen is het te starten.

---

# Uitvoeren van het programma

---

Type in the example program using a text editor, and save it. Type in het voorbeeld programma met een tekst-editor, en sla het op. Emacs is a good editor to use for this because it has its own Perl mode which formats lines nicely when you hit tab (use `Mx perl-mode'). Emacs is een goede editor te gebruiken voor dit omdat ze heeft haar eigen Perl modus die formaten lijnen mooi wanneer je hit tabblad (gebruik `Mx perl-mode '). But as ever, use whichever you're most comfortable with. Maar zoals altijd, gebruik wat je met de meest comfortabele.

After you've entered and saved the program make sure the file is executable by using the command Nadat u zich heeft opgegeven en opgeslagen het programma ervoor zorgen dat het bestand uitvoerbaar is door het gebruik van het commando

```
chmod u+x progname Chmod u + x progname
```

at the UNIX prompt, where *progname* is the filename of the program. Op de UNIX-prompt, waar *progname* is de bestandsnaam van het programma. Now to run the program just type any of the following at the prompt. Nu om het programma te gebruiken typt u een van de volgende op de prompt.

```
perl progname ./ progname progname Perl progname. / Progname progname
```

If something goes wrong then you may get error messages, or you may get nothing. Als er iets mis gaat dan kun je foutmeldingen krijgen, of je kunt krijgen niets. You can always run the program with warnings using the command U kunt altijd start het programma met waarschuwingen met behulp van het commando

```
perl -w progname Perl-w progname
```

at the prompt. Op de prompt. This will display warnings and other (hopefully) helpful messages before it tries to execute the program. Dit zal display waarschuwingen en andere (hopelijk) nuttig berichten voordat hij probeert het programma uitvoeren. To run the program with a debugger use the command Om het programma te runnen met een debugger gebruik van het commando

```
perl -d progname Perl-d progname
```

When the file is executed Perl first compiles it and then executes that compiled version. Wanneer

het bestand wordt uitgevoerd Perl eerste compileren en vervolgens voert dat gecompileerde versie. So after a short pause for compilation the program should run quite quickly. Dus na een korte pauze voor het opstellen van het programma moet draaien heel snel. This also explains why you can get compilation errors when you execute a Perl file which consists only of text. Dit verklaart ook waarom je kunt krijgen compilatie fouten voeren wanneer u een Perl-bestand die bestaat alleen uit tekst.

Make sure your program works before proceeding. Zorg ervoor dat uw programma werkt voordat u doorgaat. The program's output may be slightly unexpected - at least it isn't very pretty. De programma's output mogelijk iets onverwachte - althans het is niet erg mooi. We'll look next at variables and then tie this in with prettier printing. We zullen op de volgende variabelen en vervolgens in met dit gelijkspel mooier afgedrukt.

---

# Scalar variables Scalars

---

The most basic kind of variable in Perl is the *scalar variable* . De meest elementaire vorm van variabele in Perl is de *scalaire variabele.* Scalar variables hold both strings and numbers, and are remarkable in that strings and numbers are completely interchangable. Scalars hold beide nummers en strings, en zijn opmerkelijk dat in strings en nummers zijn volledig uitwisselbaar. For example, the statement Bijvoorbeeld, de verklaring

```
$priority = 9; $ Prioriteit = 9;
```

sets the scalar variable $priority to 9, but you can also assign a string to exactly the same variable: Stelt de scalaire variabele $ prioriteit tot en met 9, maar u kunt ook een string met precies dezelfde variabele:

```
$priority = 'high'; $ Prioriteit = 'hoog';
```

Perl also accepts numbers as strings, like this: Perl aanvaardt ook nummers als strijkers, zoals deze:

```
$priority = '9'; $default = '0009'; $ Prioriteit ='9 '; $ standaard ='0009';
```

and can still cope with arithmetic and other operations quite happily. En kan nog steeds geconfronteerd met rekenkundige en andere bewerkingen heel gelukkig.

In general variable names consists of numbers, letters and underscores, but they should not start with a number and the variable **$_** is special, as we'll see later. In het algemeen namen variabele bestaat uit cijfers, letters en underscores, maar ze moeten niet beginnen met een nummer en de variabele **$ _** speciale, zoals we later zullen zien. Also, Perl is case sensitive, so $a and $A are different. Ook Perl is hoofdlettergevoelig, dus $ a en $ A zijn verschillend.

# Operations and Assignment Operaties en Assignment

Perl uses all the usual C arithmetic operators: Perl gebruikt alle gebruikelijke C rekenkundige operatoren:

```
 $a = 1 + 2; # Add 1 and 2 and store in $a $a = 3 - 4; # Subtract 4 from 3 and store in $a $a = 5 * 6; # Multiply 5 and 6 $a = 7 / 8; # Divide 7 by 8 to give 0.875 $a = 9 ** 10; # Nine to the power of 10 $a = 5 % 2; # Remainder of 5 divided by 2 ++$a;  # Increment $a and then return it $a++;  # Return $a and then increment it --$a;  # Decrement $a and then return it $a--;  # Return $a and then decrement it $ A = 1 + 2; toevoegen # 1 en 2 en op te slaan in een $ $ a = 3-4; Aftrekken # 4 uit 3 en op te slaan in een $ $ a = 5 * 6; Vermenigvuldigen # 5 en 6 $ a = 7 / 8; Verdeel # 7 van 8 te geven 0,875 $ a = 9 ** 10; # Nine aan de macht van 10 $ a = 5% 2; # Restant van 5 gedeeld door 2 + + $ a; # Toename $ a en Vervolgens terug te $ a + +; # Return $ a en vervolgens te verhogen - $ a; # Decrement $ a en vervolgens terug te $ a -; # Return $ a en vervolgens het decrement
```

and for strings Perl has the following among others: En voor strijkers Perl heeft onder andere de volgende:

```
 $a = $b . $ A = $ b. $c; # Concatenate $b and $c $a = $bx $c; # $b repeated $c times $ C; # Concatenate $ b en $ c $ a = $ bx $ c; # $ b $ c herhaalde malen
```

To assign values Perl includes Waardes toewijzen Perl omvat

```
 $a = $b; # Assign $b to $a $a += $b; # Add $b to $a $a -= $b; # Subtract $b from $a $a .= $b; # Append $b onto $a $ A = $ b; # Wijs $ b $ a = $ a + $ b; toevoegen # $ b $ a $ a -= $ b; # Aftrekken $ b van $ a $ a .= $ b; # Append $ $ A naar b
```

Note that when Perl assigns a value with **$a = $b** it makes a copy of $b and then assigns that to $a. Merk op dat wanneer Perl kent een waarde van **$ a = $ b** maakt een kopie van $ b en dan wijst dat op $ a. Therefore the next time you change $b it will not alter $a. Vandaar dat de volgende keer dat u veranderen $ b zal niet veranderen $ a.

Other operators can be found on the **perlop** manual page. Andere operators kan worden gevonden op de **perlop** manual page. Type **man perlop** at the prompt. Type **man perlop** op de prompt.

---

# Interpolation Interpolatie

The following code prints *apples and pears* using concatenation: De volgende code print *appelen en peren* met concatenatie:

```
 $a = 'apples'; $b = 'pears'; print $a.' $ A = 'appels'; $ b =' peren '; print $
```

```
a.' and '.$b; En '. $ B;
```

It would be nicer to include only one string in the final print statement, but the line Het zou mooier zijn om alleen een string in de uiteindelijke print verklaring, maar de lijn

```
 print '$a and $b'; Print "$ a en $ b ';
```

prints literally *$a and $b* which isn't very helpful. Prints letterlijk *$ a en $ b* die is niet erg behulpzaam. Instead we can use the double quotes in place of the single quotes: In plaats daarvan kunnen we gebruik maken van de dubbele aanhalingstekens in de plaats van de enkele aanhalingstekens:

```
 print "$a and $b"; Print "$ a en $ b";
```

The double quotes force *interpolation* of any codes, including interpreting variables. De dubbele aanhalingstekens werking *interpolatie* van codes, met inbegrip van het interpreteren van variabelen. This is a much nicer than our original statement. Dit is een veel mooier dan onze oorspronkelijke verklaring. Other codes that are interpolated include special characters such as newline and tab. Andere codes die zijn geïnterpoleerd ook speciale tekens, zoals newline en tab. The code **\n** is a newline and **\t** is a tab. De code \ **n** is een newline en \ **t** is een tab.

---

# Array variabelen

---

A slightly more interesting kind of variable is the *array variable* which is a list of scalars (ie numbers and strings). Een iets meer interessante soort variabele is *een variabele* die de *array* is een lijst van scalairen (dwz getallen en strings). Array variables have the same format as scalar variables except that they are prefixed by an `@` symbol. Array variabelen hebben hetzelfde formaat als scalars behalve dat ze voorafgegaan door een `@` symbool. The statement De verklaring

```
 @food  = ("apples", "pears", "eels"); @music = ("whistle", "flute"); @ Voedsel
= ( "appels", "peren", "paling"); @ = muziek ( "whistle", "fluit");
```

assigns a three element list to the array variable @food and a two element list to the array variable @music. Kent een drie-element lijst aan de array variabele @ voedsel en een lijst met twee element aan de array variabele @ muziek.

The array is accessed by using indices starting from 0, and square brackets are used to specify the index. De array is toegankelijk met behulp van indexen vanaf 0, en vierkante haken worden gebruikt om de index. The expression De uitdrukking

```
 $food[2] $ Food [2]
```

returns *eels* . Retouren *aal.* Notice that the **@** has changed to a **$** because *eels* is a scalar. Merk op

dat de **@** is gewijzigd in een **$** omdat *paling* is een scalar.

---

# Array assignments Array opdrachten

As in all of Perl, the same expression in a different context can produce a different result. Zoals in alle Perl, dezelfde uitdrukking in een andere context kan een ander resultaat. The first assignment below explodes the @music variable so that it is equivalent to the second assignment. De eerste opdracht onder de explodeert @ muziek variabele zodat deze gelijk is aan de tweede opdracht.

```
 @moremusic = ("organ", @music, "harp"); @moremusic = ("organ", "whistle",
"flute", "harp"); @ Moremusic = ( "orgaan", @ muziek, "harp"); @ moremusic =
( "orgaan", "whistle", "fluit", "harp");
```

This should suggest a way of adding elements to an array. Dit zou wijzen op een manier van het toevoegen van elementen aan een array. A neater way of adding elements is to use the statement Een neater manier van het toevoegen van elementen is het gebruik van de verklaring

```
 push(@food, "eggs"); Push (@ voedsel, "eieren");
```

which pushes *eggs* onto the end of the array @food. Die duwt *eieren* op het einde van de array @ voedsel. To push two or more items onto the array use one of the following forms: Om twee of meer items op de array gebruik van een van de volgende vormen:

```
 push(@food, "eggs", "lard"); push(@food, ("eggs", "lard")); push(@food,
@morefood); Push (@ voedsel, "eieren", "reuzel"); push (@ voedsel, ( "eieren",
"reuzel")); push (@ voedsel, @ morefood);
```

The **push** function returns the length of the new list. De **push-functie** geeft de lengte van de nieuwe lijst.

To remove the last item from a list and return it use the **pop** function. Voor het verwijderen van het laatste item uit een lijst en terugsturen gebruik van het **pop-functie.** From our original list the **pop** function returns *eels* and @food now has two elements: Van onze oorspronkelijke lijst van de **pop-functie** geeft *de aal* en @ voedsel nu twee elementen:

```
 $grub = pop(@food); # Now $grub = "eels" $ Grub = pop (@ voedsel); # $ Nu grub
= "aal"
```

It is also possible to assign an array to a scalar variable. Het is ook mogelijk om een array naar een scalaire variabele. As usual context is important. Zoals gebruikelijk context is belangrijk. The line De lijn

```
 $f = @food; $ F = @ voedsel;
```

assigns the length of @food, but Wijst de lengte van @ voedsel, maar

```
 $f = "@food"; $ F = "@ food";
```

turns the list into a string with a space between each element. Maakt van de lijst in een string met een spatie tussen elk element. This space can be replaced by any other string by changing the value of the special **$"** variable. This variable is just one of Perl's many special variables, most of which have odd names. Deze ruimte kan worden vervangen door een andere string door het veranderen van de waarde van de speciale **$ "variabele.** Deze variabele is slechts een van de vele speciale Perl variabelen, de meeste van die rare namen.

Arrays can also be used to make multiple assignments to scalar variables: Arrays kan ook gebruikt worden om meerdere opdrachten te scalars:

```
 ($a, $b) = ($c, $d);  # Same as $a=$c; $b=$d; ($a, $b) = @food;  # $a and $b
are the first two     # items of @food. ($ A, $ b) = ($ c, $ d); # Zelfde als $
a = $ c; $ b = $ d; ($ a, $ b) = @ voedsel; # $ a en $ b zijn de Eerste twee
items van # @ voedsel. ($a, @somefood) = @food; # $a is the first item of @food
# @somefood is a list of the     # others. ($ A, @ somefood) = @ voedsel; # $ a
is het eerste item van voedsel @ @ # somefood is een lijst van de # anderen.
(@somefood, $a) = @food; # @somefood is @food and     # $a is undefined. (@
Somefood, $ a) = @ voedsel; # @ @ somefood is voedsel en # $ a is undefined.
```

The last assignment occurs because arrays are greedy, and @somefood will swallow up as much of @food as it can. De laatste toewijzing gebeurt omdat arrays zijn inhalig en @ somefood zal opslokken als veel van @ voedsel als het kan. Therefore that form is best avoided. Vandaar dat de beste vorm is vermeden.

Finally, you may want to find the index of the last element of a list. Tenslotte, wilt u misschien vinden de index van het laatste element van een lijst. To do this for the @food array use the expression Om dit te doen voor het voedsel @ array gebruik van de uitdrukking

```
 $#food $ # Voedsel
```

---

# Displaying arrays Overzicht arrays

Since context is important, it shouldn't be too surprising that the following all produce different results: Sinds context is belangrijk, het zou dus niet te verbazen dat de volgende alle verschillende resultaten:

```
 print @food; # By itself print "@food"; # Embedded in double quotes print
@food.""; # In a scalar context Print @ voedsel; # Door zelf print "@ food"; #
Embedded dubbele aanhalingstekens print @ voedsel. ""; # In een scalar context
```

---

# File handling

Here is the basic perl program which does the same as the UNIX **cat** command on a certain file. Hier is de fundamentele perl programma dat doet hetzelfde als de UNIX-commando **kat** over een bepaald bestand.

```
 #!/usr/local/bin/perl # # Program to open the password file, read it in, # print it, and close it again. #! / Usr / local / bin / perl # # Program te openen van het wachtwoord bestand, lees het in, # print, en sluit het weer. $file = '/etc/passwd';  # Name the file open(INFO, $file);  # Open the file @lines = <INFO>;  # Read it into an array close(INFO);   # Close the file print @lines;   # Print the array $ File = '/ etc / passwd'; # Naam van het bestand open (INFO, $ bestand); # Open het bestand @ lijnen = <INFO>; # Lees het in een array close (INFO); # Sluit het bestand print @ Lijnen; # Print de array
```

The **open** function opens a file for input (ie for reading). De **open** functie opent een bestand voor de ingang (dwz voor het lezen). The first parameter is the *filehandle* which allows Perl to refer to the file in future. De eerste parameter is de *filehandle* waarmee Perl te verwijzen naar het bestand in de toekomst. The second parameter is an expression denoting the filename. De tweede parameter is een aanduiding van de bestandsnaam. If the filename was given in quotes then it is taken literally without shell expansion. Als de bestandsnaam werd gegeven in aanhalingstekens dan is het letterlijk genomen zonder shell expansie. So the expression '~/notes/todolist' will not be interpreted successfully. Dus de uitdrukking '~ / notities / todolist' zal niet worden geïnterpreteerd succes. If you want to force shell expansion then use angled brackets: that is, use <~/notes/todolist> instead. Als u wilt werking shell expansie gebruik dan hoekige haakjes: dat is, gebruik <~ / notities / todolist> plaats.

The **close** function tells Perl to finish with that file. De **close** functie vertelt Perl te eindigen met dat dossier.

There are a few useful points to add to this discussion on filehandling. Er zijn een paar nuttige punten toe te voegen aan deze discussie over filehandling. First, the **open** statement can also specify a file for output and for appending as well as for input. Ten eerste, de **open** verklaring kunt ook een bestand voor de uitvoer en voor voeg evenals voor de input. To do this, prefix the filename with a **>** for output and a **>>** for appending: Om dit te doen, prefix de bestandsnaam met **een>** voor de uitgang en **een>>** voor voeg:

```
 open(INFO, $file); # Open for input open(INFO, ">$file"); # Open for output open(INFO, ">>$file"); # Open for appending open(INFO, "<$file"); # Also open for input Open (INFO, $ bestand); # Open voor input open (INFO, "> $ file"); # Open voor open output (INFO, ">> $ file"); # Open voor voeg open (INFO, "<$ File "); # Ook open voor input
```

Second, if you want to print something to a file you've already opened for output then you can use the print statement with an extra parameter. Ten tweede, indien u iets wilt afdrukken naar een

bestand dat u al hebt geopend voor de output dan kunt u gebruik maken van de verklaring print met een extra parameter. To print a string to the file with the INFO filehandle use Om het afdrukken van een string naar het bestand met de INFO filehandle gebruik

```
 print INFO "This line goes to the file.\n"; Print INFO "Deze lijn gaat naar het bestand. \ N";
```

Third, you can use the following to open the standard input (usually the keyboard) and standard output (usually the screen) respectively: Derde, kunt u gebruik maken van de volgende aan de open standaard input (meestal het toetsenbord) en standard output (meestal het scherm), respectievelijk:

```
 open(INFO, '-'); # Open standard input open(INFO, '>-'); # Open standard output
Open (INFO,'-'); # Open standaard input open (INFO,'>-'); # Open standard output
```

In the above program the information is read from a file. In het bovenstaande programma is de informatie te lezen uit een bestand. The file is the INFO file and to read from it Perl uses angled brackets. Het bestand is de INFO-bestand en het lezen van het Perl gebruikt hoekige haakjes. So the statement Dus de verklaring

```
 @lines = <INFO>; @ Lijnen = <INFO>;
```

reads the file denoted by the filehandle into the array @lines. Leest het bestand aangeduid door de filehandle in de array @ lijnen. Note that the <INFO> expression reads in the file entirely in one go. Merk op dat de <INFO> uitdrukking luidt in het dossier geheel in een keer. This because the reading takes place in the context of an array variable. Dit omdat de lezing plaatsvindt in het kader van een array variabele. If @lines is replaced by the scalar $lines then only the next one line would be read in. In either case each line is stored complete with its newline character at the end. Als @ lijnen wordt vervangen door de scalaire $ lijnen dan alleen de volgende regel zou worden gelezen inch In beide gevallen is elke regel is opgeslagen, compleet met haar newline teken aan het eind.

---

# Control structuren

---

More interesting possiblities arise when we introduce control structures and looping. Meer interessante mogelijkheden ontstaan wanneer we voeren controle structuren en looping. Perl supports lots of different kinds of control structures which tend to be like those in C, but are very similar to Pascal, too. Perl ondersteunt veel verschillende soorten controle structuren die vaak worden zoals die in C, maar zijn zeer gelijkaardig aan Pascal ook. Here we discuss a few of them. Hier bespreken we een aantal van hen.

---

# foreach Foreach

To go through each line of an array or other list-like structure (such as lines in a file) Perl uses the foreach structure. Om verder te gaan in elke regel van een array of andere lijst-achtige structuur (zoals lijnen in een bestand) Perl gebruikt de foreach structuur. This has the form Dit heeft de vorm

```
 foreach $morsel (@food)  # Visit each item in turn     # and call it $morsel {
print "$morsel\n"; # Print the item  print "Yum yum\n"; # That was nice }
Foreach $ klomp (@ food) # Bezoek elk item op zijn beurt # en noemen het klomp $
(print "$ klomp \ n"; # Print het item print "Yum yum \ n"; # Dat was leuk)
```

The actions to be performed each time are enclosed in a block of curly braces. De acties worden uitgevoerd elke keer zijn ingesloten in een blok van accolades. The first time through the block $morsel is assigned the value of the first item in the array @food. De eerste keer dat via het blok $ klomp krijgt de waarde van het eerste item in de array @ voedsel. Next time it is assigned the value of the second item, and so until the end. Volgende keer is het aangewezen de waarde van het tweede punt, en zo tot het einde. If @food is empty to start with then the block of statements is never executed. Als @ voedsel is leeg beginnen dan met het blok van de verklaringen is nooit uitgevoerd.

---

# Testing Testen

The next few structures rely on a test being true or false. De volgende paar structuren rekenen op een test worden waar of onwaar. In Perl any non-zero number and non-empty string is counted as true. In Perl enige niet-nul-nummer en niet-lege string wordt geteld als ware. The number zero, zero by itself in a string, and the empty string are counted as false. Het getal nul, nul door zelf in een string, en de lege string zijn geteld als vals. Here are some tests on numbers and strings. Hier zijn enkele tests op nummers en strings.

```
 $a == $b  # Is $a numerically equal to $b? $ A == $ b # $ Is een numeriek
gelijk aan $ b? # Beware: Don't use the = operator. # Let op: Gebruik niet de =
operator. $a != $b  # Is $a numerically unequal to $b? $ A! B = $ # $ Is een
numeriek ongelijke tot $ b? $a eq $b  # Is $a string-equal to $b? Eq $ a $ b $ #
Is een string-gelijk aan $ b? $a ne $b  # Is $a string-unequal to $b? $ A ne $ b
# $ Is een string-ongelijke tot $ b?
```

You can also use logical and, or and not: U kunt ook gebruik maken van logische en, of en niet:

```
 ($a && $b)  # Is $a and $b true? ($ A & & $ b) Is # $ a en $ b true? ($a || $b)
# Is either $a or $b true? ($ A | | $ b) # Is ofwel $ a of $ b true? !($a)   #
is $a false? ! ($ A) $ # is een vals?
```

---

# for Voor

Perl has a **for** structure that mimics that of C. It has the form Perl heeft een structuur **voor** dat eruitziet als die van C. Het heeft de vorm

```
 for ( initialise ; test ; inc ) { first_action; second_action; etc }
(Initialiseren; test; incl.) (first_action; second_action; etc)
```

First of all the statement *initialise* is executed. Allereerst de verklaring *initialiseren* is uitgevoerd. Then while *test* is true the block of actions is executed. Dan terwijl *test* is waar het blok van de acties wordt uitgevoerd. After each time the block is executed *inc* takes place. Na elke keer het blok wordt uitgevoerd *incl.* plaatsvindt. Here is an example for loop to print out the numbers 0 to 9. Hier is een voorbeeld voor de lus om te printen op de cijfers 0 tot en met 9.

```
 for ($i = 0; $i < 10; ++$i) # Start with $i = 1     # Do it while $i < 10     #
Increment $i before repeating {  print "$i\n"; } For ($ i = 0; $ i <10; + + $ i)
# Start met $ i = 1 # Do it terwijl $ i <10 # Toename $ i vóór herhalen (print
"$ i \ n";)
```

---

# while and until Terwijl tot en met

Here is a program that reads some input from the keyboard and won't continue until it is the correct password Hier is een programma dat leest enkele input van het toetsenbord en niet zal doorgaan totdat het is het juiste wachtwoord

```
 #!/usr/local/bin/perl print "Password? ";  # Ask for input $a = <STDIN>;   #
Get input chop $a;   # Remove the newline at end while ($a ne "fred")  # While
input is wrong... #! / Usr / local / bin / perl print "Wachtwoord?"; # Ask for
input $ a = <STDIN>; # Haal input hakken $ a; # Verwijder de newline aan het
eind while ($ a ne "fred") # Terwijl input is verkeerd ... {     print "sorry.
Again? "; # Ask again     $a = <STDIN>;  # Get input again     chop $a;   # Chop
off newline again } (Print "sorry. Again?"; # Vraag opnieuw $ a = <STDIN>; #
Haal input weer hakken $ a; # Hak off newline weer)
```

The curly-braced block of code is executed while the input does not equal the password. De accolades-geschoorde blok code wordt uitgevoerd, terwijl de input niet gelijk aan het wachtwoord. The **while** structure should be fairly clear, but this is the opportunity to notice several things. De structuur moet **worden, terwijl** vrij duidelijk, maar dit is de gelegenheid om mededeling meerdere dingen. First, we can we read from the standard input (the keyboard) without opening the file first. Ten eerste, we kunnen we lezen uit de standaard input (het toetsenbord) zonder opening van het bestand eerst op. Second, when the password is entered $a is given that value including the newline character at the end. Ten tweede, wanneer het wachtwoord is ingevoerd $ is een gegeven dat de waarde met inbegrip van het newline teken aan het eind. The **chop** function removes the last character of a string which in this case is the newline. De **hakken** functie verwijdert u het laatste

teken van een string die in dit geval is het newline.

To test the opposite thing we can use the **until** statement in just the same way. Om te testen het tegenovergestelde ding kunnen we gebruik maken van de verklaring **tot** in net dezelfde manier. This executes the block repeatedly until the expression is true, not while it is true. Deze voert het blok totdat de expressie is waar, maar het is niet waar.

Another useful technique is putting the **while** or **until** check at the end of the statement block rather than at the beginning. Een andere handige techniek is putting **terwijl** het controleren of **totdat** aan het einde van de verklaring blok in plaats van aan het begin. This will require the presence of the **do** operator to mark the beginning of the block and the test at the end. Dit vereist de aanwezigheid van de exploitant **doen** aan het begin van het blok en de test op het einde. If we forgo the *sorry.* Als we afzien van de *sorry. Again* message in the above password program then it could be written like this. *Nogmaals* bericht in het bovenstaande wachtwoord programma vervolgens kan worden geschreven als dit.

```
 #!/usr/local/bin/perl do {  "Password? ";  # Ask for input  $a = <STDIN>;  # Get input  chop $a;  # Chop off newline } while ($a ne "fred")  # Redo while wrong input #! / Usr / local / bin / perl doen ( "Wachtwoord?"; # Ask for input $ a = <STDIN>; # Haal input hakken $ a; # Hak off newline) while ($ a ne "fred") # Opnieuw terwijl de verkeerde ingang
```

---

# Voorwaarden

---

Of course Perl also allows if/then/else statements. Natuurlijk is het ook mogelijk als Perl / then / else statements. These are of the following form: Deze zijn van de volgende vorm:

```
 if ($a) {  print "The string is not empty\n"; } else {  print "The string is empty\n"; } If ($ a) (print "De string is niet leeg \ n";) else (print "De string is leeg \ n";)
```

For this, remember that an empty string is considered to be false. Voor deze, vergeet dan niet dat er een lege string wordt beschouwd als vals. It will also give an "empty" result if $a is the string *0* . Het zal ook een "leeg" resultaat als $ a is de string *0.*

It is also possible to include more alternatives in a conditional statement: Het is ook mogelijk om meer alternatieven in een voorwaardelijke verklaring:

```
 if (!$a)   # The ! If (! $ A) De #! is the not operator {  print "The string is empty\n"; } elsif (length($a) == 1)  # If above fails, try this {  print "The string has one character\n"; } elsif (length($a) == 2)  # If that fails, try this {  print "The string has two characters\n"; } else    # Now, everything has failed {  print "The string has lots of characters\n"; } Is de exploitant niet
```

```
(print "De string is leeg \ n";) elsif (lengte ($ a) == 1) # Als bovenstaande
niet lukt, probeer dit (print "De string is een teken \ n";) elsif (lengte ($ A)
== 2) # Als dat niet lukt, probeer dit (print "De string heeft twee tekens \
n";) else # Nu, alles is mislukt (print "De string heeft veel tekens \ n";)
```

In this, it is important to notice that the elsif statement really does have an "e" missing. In deze context is het van belang op te merken dat de elsif verklaring echt een "e" ontbreekt.

---

# String matching

---

One of the most useful features of Perl (if not *the* most useful feature) is its powerful string manipulation facilities. Een van de meest handige functies van Perl (indien niet *de* meest handige feature) is de krachtige string manipulatie faciliteiten. At the heart of this is the *regular expression* (RE) which is shared by many other UNIX utilities. In het hart van deze is de *reguliere expressie* (RE), dat wordt gedeeld door vele andere UNIX-hulpprogramma's.

---

## Regular expressions Reguliere expressies

A regular expression is contained in slashes, and matching occurs with the =~ operator. Een reguliere expressie is vervat in slashes, en matching plaatsvindt met de = ~ operator. The following expression is true if the string *the* appears in variable $sentence. De volgende expressie is waar als de string verschijnt in *de* variabele $ zin.

```
$sentence =~ /the/ $ Zin = ~ / de /
```

The RE is case sensitive, so if De RE is hoofdlettergevoelig, dus als

```
$sentence = "The quick brown fox"; $ Zin = "The quick brown fox";
```

then the above match will be false. Dan de hierboven match zullen vals. The operator **!~** is used for spotting a non-match. De **operator! ~** Wordt gebruikt voor spotting een non-match. In the above example In het bovenstaande voorbeeld

```
$sentence !~ /the/ $ Zin! ~ / De /
```

is true because the string *the* does not appear in $sentence. Waar is, omdat de string *het* niet verschijnen in $ zin.

---

# The $_ special variable De speciale variabele $ _

We could use a conditional as We kunnen gebruik maken van een voorwaardelijke als

```
 if ($sentence =~ /under/) {  print "We're talking about rugby\n"; } If ($ zin = ~ / onder /) (print "We spreken over rugby \ n";)
```

which would print out a message if we had either of the following Die zou uitprinten een bericht als we hadden een van de volgende

```
 $sentence = "Up and under"; $sentence = "Best winkles in Sunderland"; $ Zin = "Boven en onder"; $ zin = "Best winkles in Sunderland";
```

But it's often much easier if we assign the sentence to the special variable **$_** which is of course a scalar. Maar het is vaak veel eenvoudiger als we de zin van de speciale variabele **$ _** die is natuurlijk een scalar. If we do this then we can avoid using the match and non-match operators and the above can be written simply as Als we dit doen dan kunnen we met behulp van de match en niet-overeenkomen met de exploitanten en het bovenstaande kan worden geschreven als gewoon

```
 if (/under/) {  print "We're talking about rugby\n"; } If (/ onder /) (print "We spreken over rugby \ n";)
```

The **$_** variable is the default for many Perl operations and tends to be used very heavily. De **$ _** variabele is de standaard voor veel Perl operaties en de neiging om te worden gebruikt zeer zwaar.

---

# More on REs Meer over REs

In an RE there are plenty of special characters, and it is these that both give them their power and make them appear very complicated. In een RE er zijn tal van speciale tekens, en het is deze dat zowel geven ze hun macht en maken ze lijken erg ingewikkeld. It's best to build up your use of REs slowly; their creation can be something of an art form. Het is best op te bouwen is het gebruik van REs langzaam; hun creatie kunnen worden iets van een kunst.

Here are some special RE characters and their meaning Hier zijn een aantal bijzondere RE tekens en hun betekenis

```
 . # Any single character except a newline ^ # The beginning of the line or string $ # The end of the line or string * # Zero or more of the last character + # One or more of the last character ? # Een enkel teken behalve een newline ^ # Het begin van de lijn of string $ # Het einde van de lijn of string * # Zero of meer van het laatste teken + # Een of meer van de laatste karakter? # Zero or one of the last character # Zero of in een van de laatste karakter
```

and here are some example matches. En hier zijn enkele voorbeeld wedstrijden. Remember that should be enclosed in / ... / slashes to be used. Vergeet niet dat moet worden ingesloten in / ... / slashes te worden gebruikt.

```
 te # t followed by anthing followed by e  # This will match the  #
tre  #                 tle  # but not te  #         tale ^f # f at the beginning
of a line ^ftp # ftp at the beginning of a line e$ # e at the end of a line tle$
# tle at the end of a line und* # un followed by zero or more d characters  #
This will match un  #                 und  #                 undd  #
unddd (etc) .* # Any string without a newline. Te # t, gevolgd door anthing
gevolgd door e # Dit zal overeenkomen met de tre # # # tle maar niet te sprookje
^ # f # f aan het begin van een regel ^ ftp ftp # aan het begin van een regel e
$ # e op de Einde van een regel tle $ # tle aan het einde van een regel und * #
un gevolgd door nul of meer tekens d # Dit zal match un # # und undd # unddd
(etc) .* # Elke string zonder een newline. This is because  # the . Dit komt
omdat de #. matches anything except a newline and  # the * means zero or more of
these. Wedstrijden alles behalve een newline en de # * betekent nul of meer van
deze. ^$ # A line with nothing in it. ^ $ # A lijn met de in het niets.
```

There are even more options. Er zijn nog meer opties. Square brackets are used to match any one of the characters inside them. Vierkante haakjes worden gebruikt voor de match op een van de karakters in hen. Inside square brackets a **-** indicates "between" and a ^ at the beginning means "not": Binnen vierkante haken een **-** geeft "tussen" en een ^ aan het begin betekent "niet":

```
 [qjk]  # Either q or j or k [^qjk]  # Neither q nor j nor k [az]  # Anything
from a to z inclusive [^az]  # No lower case letters [a-zA-Z] # Any letter [az]+
# Any non-zero sequence of lower case letters [Qjk] # Ofwel q of j of k [^ qjk]
# Noch q noch noch j k [az] # Alles van a tot z inclusieve [^ az] # Geen kleine
letters [a-zA-Z] # Elke brief [Az] + # Elke non-zero opeenvolging van letters
```

At this point you can probably skip to the end and do at least most of the exercise. Op dit punt kan je waarschijnlijk naar het einde en doen tenminste de meeste van de oefening. The rest is mostly just for reference. De rest is meestal alleen voor de verwijzing.

A vertical bar **|** represents an "or" and parentheses **(** ... **)** can be used to group things together: Een verticale balk **|** vertegenwoordigt een "of" en tussen haakjes **(...)** kan worden gebruikt om groep samen dingen:

```
 jelly|cream # Either jelly or cream (eg|le)gs # Either eggs or legs (da)+  #
Either da or dada or dadada or... Gelei | room # Ofwel gelei of crème (bv. | le)
gs # Ofwel eieren of benen (da) + # Ofwel da of dada of dadada of ...
```

Here are some more special characters: Hier zijn nog enkele speciale tekens:

```
 \n  # A newline \t  # A tab \w  # Any alphanumeric (word) character. \ N # A
newline \ t # A tabblad \ w # Elke alfanumerieke (woord) karakter. # The same as
[a-zA-Z0-9_] \W  # Any non-word character. # De hetzelfde als [a-zA-Z0-9_] \ W #
Elke niet-woord karakter. # The same as [^a-zA-Z0-9_] \d  # Any digit. # De
hetzelfde als [^ a-zA-Z0-9_] \ d # Elk cijfer. The same as [0-9] \D  # Any non-
digit. De hetzelfde als [0-9] \ D # Elke niet-cijfer. The same as [^0-9] \s  #
Any whitespace character: space,   # tab, newline, etc \S  # Any non-whitespace
character \b  # A word boundary, outside [] only \B  # No word boundary De
hetzelfde als [^ 0-9] \ s # Elke spatieteken: ruimte, # tab, newline, etc \ S #
```

```
Elke niet-scheidingsteken \ b # A woord grens, buiten alleen [] \ B # Geen woord boundary
```

Clearly characters like **$** , | , **[** , **)** , \ , / and so on are peculiar cases in regular expressions. Duidelijk tekens zoals **$, | [,), \,** / en dus op zijn bijzondere gevallen in reguliere expressies. If you want to match for one of those then you have to preceed it by a backslash. Als u wilt match voor een van deze dan moet je voorafgaan door een backslash. So: Dus:

```
 \|  # Vertical bar \[  # An open square bracket \)  # A closing parenthesis \* # An asterisk \^  # A carat symbol \/  # A slash \\  # A backslash \ | # Vertical bar \ [# Een open vierkante beugel \) # Een sluitende haakjes \ * # Een sterretje \ ^ # A karaat symbool \ / # A slash \ \ # A backslash
```

and so on. Enzovoort.

---

# Vervanging en vertaling

---

As well as identifying regular expressions Perl can make substitutions based on those matches. Alsook het identificeren van Perl reguliere expressies kunnen maken substituties op basis van die wedstrijden. The way to do this is to use the **s** function which is designed to mimic the way substitution is done in the vi text editor. De manier om dit te doen is met behulp van de functie **en** die is ontworpen om te lijken op de manier waarop vervanging is gedaan in de vi text editor. Once again the match operator is used, and once again if it is omitted then the substitution is assumed to take place with the **$_** variable. Nogmaals de match operator wordt gebruikt, en nogmaals, indien deze niet wordt uitgegaan van de vervanging dient plaats te vinden met de **$ _** variabele.

To replace an occurrence of *london* by *London* in the string $sentence we use the expression Ter vervanging van een optreden van *london* door *Londen* in de string $ zin gebruiken we de uitdrukking

```
$sentence =~ s/london/London/ $ Zin = ~ s / london / Londen /
```

and to do the same thing with the **$_** variable just En hetzelfde te doen met de **$ _** variabele gewoon

```
s/london/London/ S / london / Londen /
```

Notice that the two regular expressions ( *london* and *London* ) are surrounded by a total of three slashes. Merk op dat de twee reguliere expressies *(london* en *Londen)* zijn omgeven door een totaal van drie streepjes. The result of this expression is the number of substitutions made, so it is either 0 (false) or 1 (true) in this case. Het resultaat van deze uitdrukking is het aantal vervangingen gedaan, dus het is ofwel 0 (false) of 1 (waar) in dit geval.

---

# Options Opties

This example only replaces the first occurrence of the string, and it may be that there will be more than one such string we want to replace. Dit voorbeeld alleen het eerste voorkomen van de string, en het zou kunnen dat er meer dan een string die we willen vervangen. To make a global substitution the last slash is followed by a **g** as follows: Om een globale substitutie de laatste slash wordt gevolgd door een **g** als volgt:

```
 s/london/London/g S / london / Londen / g
```

which of course works on the **$_** variable. Die natuurlijk werkt op de **$ _** variabele. Again the expression returns the number of substitutions made, which is 0 (false) or something greater than 0 (true). Nogmaals de uitdrukking geeft het aantal substituties gemaakt, dat is 0 (false), of iets meer dan 0 (true).

If we want to also replace occurrences of *lOndon* , *lonDON* , *LoNDoN* and so on then we could use Als we willen ook vervangen gebeurtenissen van *lOndon, lonDON, LoNDoN* en dus op dan we kunnen gebruiken

```
 s/[Ll][Oo][Nn][Dd][Oo][Nn]/London/g S / [Ll] [Oo] [Nn] [D quinquies] [Oo] [Nn]
/ London / g
```

but an easier way is to use the **i** option (for "ignore case"). Maar een gemakkelijker manier is het gebruik van de **i-optie** (voor "negeren geval"). The expression De uitdrukking

```
 s/london/London/gi S / london / Londen / gi
```

will make a global substitution ignoring case. Zal een mondiale substitutie negeren geval. The **i** option is also used in the basic / ... / regular expression match. De **i** optie wordt ook gebruikt in de fundamentele / ... / reguliere expressie match.

---

# Remembering patterns Herinneren patronen

It's often useful to remember patterns that have been matched so that they can be used again. Het is vaak handig om te onthouden patronen die zijn afgestemd, zodat ze kunnen worden gebruikt. It just so happens that anything matched in parentheses gets remembered in the variables **$1** ,..., **$9** . Het gebeurt dat net zo iets gevonden tussen haakjes wordt herdacht in de variabelen **$ 1** ,..., **$ 9.** These strings can also be used in the same regular expression (or substitution) by using the special RE codes **\1** ,..., **\9** . Deze reeksen kunnen ook worden gebruikt in dezelfde reguliere expressie (of vervanging) door gebruik te maken van de speciale codes RE **\ 1** ,..., **\ 9.** For example Bijvoorbeeld

```
 $_ = "Lord Whopper of Fibbing"; s/([AZ])/:\1:/g; print "$_\n"; $ _ = "Heer
Whopper van Fibbing"; s / ([AZ]) /: \ 1: / g; print "$ _ \ n";
```

will replace each upper case letter by that letter surrounded by colons. Komt in de plaats van elk

hoofdletter door die brief omringd door dubbele punten. It will print *:L:ord :W:hopper of :F:ibbing* . Het zal *print: L: ord: W: van hopper: F: ibbing.* The variables **$1** ,..., **$9** are read-only variables; you cannot alter them yourself. De variabelen **$ 1** ,..., **$ 9** zijn alleen-lezen variabelen; je niet kan veranderen ze zelf.

As another example, the test Een ander voorbeeld: de proef

```
 if (/(\b.+\b) \1/) {  print "Found $1 repeated\n"; } If (/ (\ b. + \ b) \ 1 /)
(print "Gevonden $ 1 herhaald \ n";)
```

will identify any words repeated. Zal vast welke woorden herhaalde. Each **\b** represents a word boundary and the **.+** matches any non-empty string, so **\b.+\b** matches anything between two word boundaries. Elke \ **b** vertegenwoordigt een woord en de **grens.** + Komt overeen met een niet-lege string, zo \ **b.** + \ **b** wedstrijden iets tussen twee woord grenzen. This is then remembered by the parentheses and stored as **\1** for regular expressions and as **$1** for the rest of the program. Dit wordt vervolgens gewezen door de haakjes en opgeslagen als \ **1** voor reguliere expressies en **$ 1** voor de rest van het programma.

The following swaps the first and last characters of a line in the **$_** variable: De volgende swaps de eerste en de laatste letters van een regel in de **$ _** variabele:

```
 s/^(.)(.*)(.)$/\3\2\1/ S /^(.)(.*)(.)$/ \ 3 \ 2 \ 1 /
```

The ^ and **$** match the beginning and end of the line. De ^ en **$** match het begin en het einde van de regel. The **\1** code stores the first character; the **\2** code stores everything else up the last character which is stored in the **\3** code. De \ **1** code winkels het eerste teken; de \ **2** code slaat alles op het laatste karakter dat wordt opgeslagen in de \ **3** code. Then that whole line is replaced with **\1** and **\3** swapped round. Dan is dat hele lijn is vervangen met \ **1** en \ **3** swap ronde.

After a match, you can use the special read-only variables **$`** and **$&** and **$'** to find what was matched before, during and after the seach. Na een wedstrijd, kunt u gebruik maken van de speciale read-only variabelen **$ `en $ &** en **$ 'te** vinden wat werd geëvenaard vóór, tijdens en na de zoekactie. So after Dus na

```
 $_ = "Lord Whopper of Fibbing"; /pp/; $ _ = "Heer Whopper van Fibbing"; / pp /;
```

all of the following are true. Alle van de volgende kloppen. (Remember that eq is the string-equality test.) (Vergeet niet dat eq is de string-gelijkheid test.)

```
 $` eq "Lord Wo"; $& eq "pp"; $' eq "er of Fibbing"; $ `Eq" Heer Wo "; $ & eq"
pp "; $ 'eq" eh van Fibbing ";
```

Finally on the subject of remembering patterns it's worth knowing that inside of the slashes of a match or a substitution variables are interpolated. Tot slot over het onderwerp van de herdenking van patronen is het de moeite waard te weten dat de binnenkant van de streepjes van een wedstrijd of een vervanging variabelen zijn geïnterpoleerd. So Dus

```
 $search = "the"; s/$search/xxx/g; $ Search = ""; s / $ search / xxx / g;
```

will replace every occurrence of *the* with *xxx* . Komt in de plaats van elk optreden van *de* met *xxx.* If you want to replace every occurence of *there* then you cannot do **s/$searchre/xxx/** because this will be interpolated as the variable $searchre. Als u wilt vervangen elke keer dat *er* dan kun je niet doen **s / $ searchre / xxx /** omdat deze zal worden geïnterpoleerd als de variabele $ searchre. Instead you should put the variable name in curly braces so that the code becomes In plaats daarvan moet je de naam van de variabele in accolades, zodat de code wordt

```
$search = "the"; s/${search}re/xxx/; $ Search = ""; s / $ () zoek opnieuw / xxx /;
```

---

## Translation Vertaling

The **tr** function allows character-by-character translation. De **tr** functie kunt teken-voor-teken vertaling. The following expression replaces each *a* with *e* , each *b* with *d* , and each *c* with *f* in the variable $sentence. De volgende expressie komt in de plaats van elk met *een e,* elk met *d b, c* en elk met *f* in de variabele $ zin. The expression returns the number of substitutions made. De uitdrukking geeft het aantal vervangingen gedaan.

```
$sentence =~ tr/abc/edf/ $ Zin = ~ tr / abc / edf /
```

Most of the special RE codes do not apply in the tr function. De meeste van de speciale RE-codes zijn niet van toepassing in de tr functie. For example, the statement here counts the number of asterisks in the $sentence variable and stores that in the $count variable. Bijvoorbeeld, de verklaring hier telt het aantal sterretjes in de zin $ variabele en winkels die in de variabele $ count.

```
$count = ($sentence =~ tr/*/*/); $ Count = ($ zin = ~ tr /*/*/);
```

However, the dash is still used to mean "between". Echter, het dashboard is nog steeds gebruikt in de betekenis "tussen". This statement converts **$_** to upper case. Deze verklaring zet **$ _** om naar hoofdletters.

```
tr/az/AZ/; Tr / az / AZ /;
```

---

# Split

---

A very useful function in Perl is **split** , which splits up a string and places it into an array. Een zeer nuttige functie in Perl wordt **gesplitst,** die splitst een string en plaatst het in een array. The function uses a regular expression and as usual works on the **$_** variable unless otherwise specified. De functie maakt gebruik van een reguliere expressie en zoals gewoonlijk werkt op de **$ _** variabele tenzij anders vermeld.

The **split** function is used like this: De **gesplitste** functie wordt gebruikt als volgt:

```
 $info = "Caine:Michael:Actor:14, Leafy Drive"; @personal = split(/:/, $info); $
Info = "Caine: Michael: Acteur: 14, Leafy Drive"; persoonlijke @ = split (/:/, $
info);
```

which has the same overall effect as Die heeft hetzelfde effect als

```
 @personal = ("Caine", "Michael", "Actor", "14, Leafy Drive"); @ Persoonlijke =
( "Caine", "Michael", "Actor", "14, Leafy Drive");
```

If we have the information stored in the **$_** variable then we can just use this instead Als we de informatie die is opgeslagen in de **$ _** variabele dan kunnen we alleen gebruik maken van deze plaats

```
 @personal = split(/:/); @ Persoonlijke = split (/:/);
```

If the fields are divided by any number of colons then we can use the RE codes to get round this. Als de velden worden gedeeld door een aantal van de darm, dan kunnen we gebruik maken van de RE codes om deze ronde. The code De code

```
 $_ = "Capes:Geoff::Shot putter:::Big Avenue"; @personal = split(/:+/); $ _ =
"Kapen: Geoff:: Shot putter::: Big Avenue"; persoonlijke @ = split (/:+/);
```

is the same as Is hetzelfde als

```
 @personal = ("Capes", "Geoff",              "Shot putter", "Big Avenue"); @
Persoonlijke = ( "Kapen", "Geoff", "Shot putter", "Big Avenue");
```

But this: Maar dit:

```
 $_ = "Capes:Geoff::Shot putter:::Big Avenue"; @personal = split(/:/); $ _ =
"Kapen: Geoff:: Shot putter::: Big Avenue"; persoonlijke @ = split (/:/);
```

would be like Zou graag

```
 @personal = ("Capes", "Geoff", "",              "Shot putter", "", "", "Big
Avenue"); @ Persoonlijke = ( "Kapen", "Geoff", "", "Shot putter", "", "", "Big
Avenue");
```

A word can be split into characters, a sentence split into words and a paragraph split into sentences: Een woord kan worden opgesplitst in karakters, een zin gesplitst in woorden en een paragraaf gesplitst in zinnen:

```
 @chars = split(//, $word); @words = split(/ /, $sentence); @sentences =
split(/\./, $paragraph); @ Chars = split (/ /, $ woord); @ woorden = split (/ /,
$ zin); zinnen @ = split (/ \. / $ Paragraaf);
```

In the first case the null string is matched between each character, and that is why the @chars array is an array of characters - ie an array of strings of length 1. In het eerste geval is de null string is gevonden tussen elk karakter, en dat is de reden waarom de @ chars array is een array van karakters

- dwz een array van strings van lengte 1.

---

# Associatieve arrays

---

Ordinary list arrays allow us to access their element by number. Gewone lijst arrays stellen ons in staat om toegang te krijgen tot hun element van het totale aantal. The first element of array @food is $food[0]. Het eerste element van de array @ voedsel is $ food [0]. The second element is $food[1], and so on. Het tweede element is $ levensmiddelen [1], enzovoort. But Perl also allows us to create arrays which are accessed by string. Maar Perl geeft ons ook de mogelijkheid te creëren arrays die zijn benaderd door string. These are called *associative arrays* . Dit zijn de zogenaamde *associatieve arrays.*

To define an associative array we use the usual parenthesis notation, but the array itself is prefixed by a % sign. Definiëren van een associative array gebruiken we de gebruikelijke notatie haakjes, maar de array zelf is voorafgegaan door een% teken. Suppose we want to create an array of people and their ages. Stel we willen maken van een array van mensen en hun leeftijden. It would look like this: Het zou er als volgt uitzien:

```
 %ages = ("Michael Caine", 39,          "Dirty Den", 34,          "Angie", 27,
"Willy", "21 in dog years",          "The Queen Mother", 108); % Leeftijden =
( "Michael Caine", 39, "Dirty Den", 34, "Angie", 27, "Willy", "21 jaar in hond",
"The Queen Mother", 108);
```

Now we can find the age of people with the following expressions Nu kunnen we de leeftijd van mensen met de volgende uitdrukkingen

```
 $ages{"Michael Caine"};  # Returns 39 $ages{"Dirty Den"};  # Returns 34
$ages{"Angie"};   # Returns 27 $ages{"Willy"};   # Returns "21 in dog years"
$ages{"The Queen Mother"}; # Returns 108 $ Leeftijden ( "Michael Caine");
Returns # 39 $ leeftijden ( "Dirty Den"); Returns # 34 $ leeftijden ( "Angie");
Returns # 27 $ leeftijden ( "Willy"); # Returns "21 jaar in hond "$ (Leeftijden"
The Queen Mother "); Returns # 108
```

Notice that like list arrays each **%** sign has changed to a **$** to access an individual element because that element is a scalar. Merk op dat als lijst arrays **elk%** teken is gewijzigd in een **$** toegang tot een individueel element want dat element is een scalar. Unlike list arrays the index (in this case the person's name) is enclosed in curly braces, the idea being that associative arrays are fancier than list arrays. In tegenstelling tot de lijst arrays de index (in dit geval de naam van de persoon) is bijgevoegd in accolades, het idee is dat associatieve arrays zijn liefhebber dan lijst arrays.

An associative array can be converted back into a list array just by assigning it to a list array variable. Een associatieve array kan worden geconverteerd terug in een lijst array alleen door het toewijzen aan een lijst array variabele. A list array can be converted into an associative array by

assigning it to an associative array variable. Een lijst array kan worden omgezet in een associatieve array door het toewijzen aan een associatieve array variabele. Ideally the list array will have an even number of elements: Idealiter de lijst array zal hebben zelfs een aantal elementen:

```
 @info = %ages;  # @info is a list array. Info @ =% leeftijden; # @ info is een lijst array. It    # now has 10 elements $info[5];  # Returns the value 27 from # the list array @info %moreages = @info; # %moreages is an associative    # array. Het # nu heeft 10 elementen $ info [5]; # Geeft de waarde 27 uit de lijst array # @ info% moreages = @ info; #% moreages is een associatieve # array. It is the same as %ages Het is hetzelfde als% leeftijden
```

---

# Operators Operators

Associative arrays do not have any order to their elements (they are just like hash tables) but is it possible to access all the elements in turn using the **keys** function and the **values** function: Associatieve arrays hebben geen oog op hun elementen (ze zijn net zoals hash tabellen), maar is het mogelijk om toegang te krijgen tot alle elementen op zijn beurt met behulp van de **toetsen** en de **waarden** functie:

```
 foreach $person (keys %ages) {  print "I know the age of $person\n"; } foreach $age (values %ages) {  print "Somebody is $age\n"; } Foreach $ persoon (toetsen% leeftijden) (print "Ik weet dat de leeftijd van de persoon $ \ n";) foreach $ leeftijd (waarden% leeftijden) (print "Iemand is $ leeftijd \ n";)
```

When **keys** is called it returns a list of the keys (indices) of the associative array. Wanneer **toetsen** heet geeft hij een overzicht van de sleutels (indices) van de associative array. When **values** is called it returns a list of the values of the array. Wanneer **waarden** heet geeft hij een overzicht van de waarden van de array. These functions return their lists in the same order, but this order has nothing to do with the order in which the elements have been entered. Deze functies terug in hun lijsten in dezelfde volgorde, maar deze order heeft niets te maken met de volgorde waarin de elementen zijn opgenomen.

When **keys** and **values** are called in a scalar context they return the number of key/value pairs in the associative array. Wanneer **de sleutels** en **waarden** zijn genoemd in een scalar context zij terugkeer van het aantal sleutel / waarde-paren in de associative array.

There is also a function **each** which returns a two element list of a key and its value. Er is ook een functie die **elke** twee element geeft een lijst van een toets en de waarde ervan. Every time **each** is called it returns another key/value pair: **Elke** keer wordt genoemd geeft hij een andere sleutel / waarde pair:

```
 while (($person, $age) = each(%ages)) {  print "$person is $age\n"; } While (($ persoon, $ leeftijd) = elk (% leeftijden)) (print "$ persoon is $ leeftijd \ n";)
```

---

## Environment variables Omgevingsvariabelen

When you run a perl program, or any script in UNIX, there will be certain environment variables set. Als u een perl programma, of een script in UNIX, zullen er bepaalde omgevingsvariabelen ingesteld. These will be things like USER which contains your username and DISPLAY which specifies which screen your graphics will go to. Deze zullen worden dingen als GEBRUIKER bevat uw gebruikersnaam en DISPLAY die specificeert welke uw grafische scherm zal gaan. When you run a perl CGI script on the World Wide Web there are environment variables which hold other useful information. Als u een perl CGI-script op het World Wide Web zijn er omgevingsvariabelen die in het bezit zijn andere nuttige informatie. All these variables and their values are stored in the associative **%ENV** array in which the keys are the variable names. Al deze variabelen en hun waarden worden opgeslagen in de **associatieve% ENV** array waarin de toetsen zijn de namen van variabelen. Try the following in a perl program: Probeer de volgende in een perl programma:

```
 print "You are called $ENV{'USER'} and you are "; print "using display
$ENV{'DISPLAY'}\n"; Print "U bent $ ENV ( 'USER') en u bent"; print "met display
$ ENV ( 'DISPLAY') \ n";
```

---

# Subroutines

---

Like any good programming langauge Perl allows the user to define their own functions, called *subroutines* . Zoals elke goede programmering taal Perl kan de gebruiker bij het definiëren van hun eigen functies, de zogenaamde *subroutines.* They may be placed anywhere in your program but it's probably best to put them all at the beginning or all at the end. Ze kunnen overal geplaatst worden in uw programma, maar het is waarschijnlijk het beste om ze allemaal aan het begin of aan het einde alle. A subroutine has the form Een subroutine heeft de vorm

```
 sub mysubroutine {  print "Not a very interesting routine\n";  print "This does
the same thing every time\n"; } Sub mysubroutine (print "Niet een zeer
interessante routine \ n"; print "Dit doet hetzelfde elke keer \ n";)
```

regardless of any parameters that we may want to pass to it. Ongeacht de parameters die we willen doorgeven aan. All of the following will work to call this subroutine. Alle van de volgende zal werken aan de oproep deze subroutine. Notice that a subroutine is called with an **&** character in front of the name: Merk op dat een subroutine wordt aangeroepen met een **&** teken voor de naam:

```
 &mysubroutine;  # Call the subroutine &mysubroutine($_); # Call it with a
parameter &mysubroutine(1+2, $_); # Call it with two parameters & Mysubroutine;
# Bel subroutine & mysubroutine ($_); # Bel met een parameter & mysubroutine (1
+2, $ _); # Noem het met twee parameters
```

## Parameters Parameters

In the above case the parameters are acceptable but ignored. In het bovenstaande geval de parameters zijn aanvaardbaar, maar genegeerd. When the subroutine is called any parameters are passed as a list in the special **@_** list array variable. Wanneer de subroutine heet alle parameters die worden doorgegeven als een lijst in de speciale **@ _** lijst array variabele. This variable has absolutely nothing to do with the **$_** scalar variable. Deze variabele heeft absoluut niets te maken met de **$ _** scalaire variabele. The following subroutine merely prints out the list that it was called with. De volgende subroutine alleen prenten uit de lijst dat hij werd opgeroepen. It is followed by a couple of examples of its use. Het wordt gevolgd door een paar voorbeelden van het gebruik ervan.

```
 sub printargs {  print "@_\n"; }  &printargs("perly", "king"); # Example prints "perly king" &printargs("frog", "and", "toad"); # Prints "frog and toad" Sub printargs (print "@ _ \ n";) & printargs ( "perly", "koning"); # Voorbeeld afdrukken "perly koning" & printargs ( "kikker", "en", "pad"); # Prints "kikker en Pad "
```

Just like any other list array the individual elements of **@_** can be accessed with the square bracket notation: Net als alle andere lijst array de afzonderlijke elementen van **@ _** kan worden benaderd met de vierkante beugel notatie:

```
 sub printfirsttwo {  print "Your first argument was $_[0]\n";  print "and $_[1] was your second\n"; } Sub printfirsttwo (print "Uw eerste argument was $ _ [0] \ n"; print "en $ _ [1] was uw tweede \ n";)
```

Again it should be stressed that the indexed scalars **$_[0]** and **$_[1]** and so on have nothing to with the scalar **$_** which can also be used without fear of a clash. Nogmaals zij erop gewezen dat de geïndexeerde scalairen **$ _ [0]** en **$ _ [1],** enzovoort hebben niets aan met de scalaire **$ _** die ook kan worden gebruikt zonder angst voor een botsing.

---

## Returning values Terugkerende waarden

Result of a subroutine is always the last thing evaluated. Uitslag van een subroutine is altijd het laatste wat geëvalueerd. This subroutine returns the maximum of two input parameters. Deze subroutine geeft het maximum van twee input parameters. An example of its use follows. Een voorbeeld van het gebruik ervan volgt.

```
 sub maximum {  if ($_[0] > $_[1])  {   $_[0];  }  else  {   $_[1];  } } $biggest = &maximum(37, 24); # Now $biggest is 37 Sub maximum (if ($ _ [0]> $ _ [1]) ($ _ [0];) else ($ _ [1];)) $ grootste = & maximum (37, 24); # $ grootste Nu is 37
```

The &printfirsttwo subroutine above also returns a value, in this case 1. De & printfirsttwo

subroutine boven ook terug een waarde, in dit geval 1. This is because the last thing that subroutine did was a **print** statement and the result of a successful **print** statement is always 1. Dit is omdat de laatste ding dat subroutine deden was een **print** verklaring en het resultaat van een geslaagde **afdruk** verklaring is altijd 1.

---

# Local variables Lokale variabelen

The **@_** variable is local to the current subroutine, and so of course are **$_[0]** , **$_[1]** , **$_[2]** , and so on. De **@ _** lokale variabele is in de huidige subroutine, en dus uiteraard zijn **$ _ [0], $ _ [1], $ _ [2],** enzovoort. Other variables can be made local too, and this is useful if we want to start altering the input parameters. Andere variabelen kunnen worden gemaakt lokale ook, en dit is handig als we willen starten, tot wijziging van de input parameters. The following subroutine tests to see if one string is inside another, spaces not withstanding. De volgende subroutine tests om te zien of een string zich in een ander, spaties niet tegen. An example follows. Een voorbeeld volgt.

```
 sub inside {  local($a, $b);   # Make local variables  ($a, $b) = ($_[0], $_[1]); # Assign values  $a =~ s/ //g;   # Strip spaces from  $b =~ s/ //g;   # local variables  ($a =~ /$b/ || $b =~ /$a/); # Is $b inside $a      #   or $a inside $b? Subsectoren binnen (lokale ($ a, $ b); # Zorg lokale variabelen ($ a, $ b) = ($ _ [0], $ _ [1]); # Wijs waarden $ a = ~ s / / / g ; # Strip ruimten van $ b = ~ s / / / g; # lokale variabelen ($ a = ~ / $ b / | | $ b = ~ / $ a /); # Is $ $ b binnen een # of $ a Binnen $ b? }  &inside("lemon", "dole money");  # true ) & Binnenkant ( "citroen", "dole geld"); # true
```

In fact, it can even be tidied up by replacing the first two lines with Sterker nog, het kan zelfs worden de boel door vervanging van de eerste twee lijnen met

```
 local($a, $b) = ($_[0], $_[1]); Lokale ($ a, $ b) = ($ _ [0], $ _ [1]);
```

---