# Perl Beginners' Site

Perl - because programming should be fun.

Home → Online Tutorials → The "Perl for Newbies" Tutorial → Part 2

- About Us
- Contact

- Home
- About
- News
- Links
- Perl Humour

## Resources

- Online Tutorials
  - Modern Perl by chromatic
  - The "Perl for Newbies" Tutorial
    - Part 1
    - Part 2
    - Part 3
    - Part 4
    - Part 5
  - Impatient Perl
  - Hyperpolyglot
    - Sheet 1
    - Sheet 2
  - Elements to Avoid
  - In Other Languages
- Books
  - Advanced Books
  - Topic-related Books
- IDEs and Development Tools
  - From perl.net.au
- Core Docs
- Article Collections
- Training
- FAQs
  - Freenode's #perl FAQ
    - Freenode's #perlcafe
- Exercises and Challenges

- Mailing Lists
- Web Forums
- IRC Channels

- Reference Resources
- Wikis
- Blogs

## Platforms

- Mac OS
- UNIX/Linux
- Windows

## Common Uses

- Bio-Info
- Chat Bots and Scripting (IRC, XMPP)
- Databases
- Email
- Games and Multimedia
- GUI Development
- Multitasking and Networking
- QA and Testing
- SSH/Telnet
- Sys Admin
- Text Generation
- Text Parsing
- Web Automation
- Web/CGI
- XML

## Perl Topics

- Date and Time
- Debugging
- Files and Directories
- Hashes
- Modules and Packages
- References
- Regular Expressions
- Object Oriented Perl
- Optimising and Profiling
- Security
    - Code/Markup Injection
- Scoping and Variables
- Using CPAN
    - CPAN Wrappers for Creating System Packages
    - Finding Stuff on CPAN

## Advocacy

- What about Perl 6?
- "Perl", and "perl", but not "PERL"
- Get a Job!
- Why Perl is Good
- Who is Using Perl?

## Site Resources

## Contribute

- Contributors List
- Site's Source Code

# "Perl for Newbies" - Part 2 - The Perl Beginners' Site

Learn Perl Now!
And get a job doing Perl.

Show Navigation Controls

## "Perl for Perl Newbies" - Part 2 ¶

## Contents ¶

## Licence ¶

## 1. The for Loop ¶

The for loop has another syntax which can be used to encapsulate while-like loops in a more compact way, which also has some advantages.

The syntax for this for loop is for({Initial Statements} ; {Condition} ; {Iteration Commands}) { ... }, where {Condition} is a boolean condition expression, and {Initial Statements} and {Iteration Commands} may contain one or more commands separated by commas.

When The interpreter encounters a for loop it executes its initial statements, and then executes the commands within its block for as long as the condition is met. {Iteration Commands} are executed at the end of each iteration.

The following simple example prints the multiplication board:

```
for($row = 1 ; $row <= 10 ; $row++)
{
    for($column = 1 ; $column <= 10 ; $column++)
    {
        $result = $row*$column;
        $pad = ( " "  x  (4-length($result)) );
        print $pad, $result;
    }
    print "\n";
}
```

## 1.1. Behaviour of next in the for Loop ¶

An important difference between the for loop and the while loop is the behaviour of next inside it. In a for loop, next skips the rest of the iteration but still executes the iteration commands which were passed to the loop as arguments.

For example, the following program prints the first 200 primes:

```
MAIN_LOOP: for(
    @primes=(2), $i=3 ;
    scalar(@primes) < 200 ;
    $i++
    )
{
    foreach $p (@primes)
    {
        if ($i % $p == 0)
        {
            next MAIN_LOOP;
        }
    }

    push @primes, $i;
}

print join(", " , @primes), "\n";
```

Now, the equivalent program using a while loop:

```
@primes = (2);
$i = 3;
MAIN_LOOP:
while (scalar(@primes) < 200)
{
    foreach $p (@primes)
    {
        if ($i % $p == 0)
        {
            next MAIN_LOOP;
        }
    }

    push @primes, $i;
    $i++;
}

print join(", " , @primes), "\n";
```

gets stuck. (**why?**)

## 1.2. Whence for? ¶

So far we have learned about three different kinds of for loops, so you are probably wondering when to use each one.

A substantial difference between the notation `for($i=0 ; $i<$limit ; $i++)` and between `for $i (0 .. ($limit-1))` is that the first accommodates itself to changes in `$limit`, while the second will iterate for a constant number of times. Of course, you can make the first one similar to the second by assigning the value of `$limit` to a third variable that will remain constant. Still, the second notation is usually safer from that sense.

The `foreach $item (@array)` loop is quite handy, but sometimes when iterating over an array of items the index of the item is of importance. In that case, one may prefer to use a normal `for` loop.

## 2. Hashes ¶

Hashes can be used to map a set of keys, each to his own value. Using a hash one can retrieve the value associated with each key, as well as get a list of all the keys present in the hash.

To assign or retrieve the value of the key `$mykey` in the hash `$myhash` one uses the `$myhash{$mykey}` convention. To check if a key exists in a hash one should type `exists($myhash{$mykey})` which in turn returns a boolean value that corresponds to its existence.

An array whose elements are the keys present in the hash can be retrieved by typing `keys(%myhash)`. Here's a short example, that runs a simple bank, that will illustrate this functionality:

```
# Initialize the valid operations collection
$ops{'create'} = 1;
$ops{'deposit'} = 1;
$ops{'status'} = 1;
$ops{'exit'} = 2;

while (1)
{
    # Get a valid input from the user.
    while (1)
    {
        print "Please enter what you want to do:\n";
        print "(" . join(", ", keys(%ops)) . ")\n";

        $function = <>;
        chomp($function);
        if (exists($ops{$function}))
        {
            last;
        }
        print "Unknown function!\n Please try again.\n\n"
    }

    if ($function eq "exit")
    {
        last;
    }

    print "Enter the name of the account:\n";
    $account = <>;
    chomp($account);
    if ($function eq "create")
    {
        if (! exists($bank_accounts{$account}))
        {
            $bank_accounts{$account} = 0;
        }
    }
    elsif ($function eq "status")
    {
        if (! exists ($bank_accounts{$account}) )
        {
            print "Error! The account does not exist.\n";
        }
        else
        {
            print "There are " . $bank_accounts{$account} .
                " NIS in the account.\n";
        }
    }
    elsif ($function eq "deposit")
    {
        if (exists($bank_accounts{$account}) )
        {
            print "How much do you wish to deposit?\n";
            $how_much = <>;
            chomp($how_much);
            $bank_accounts{$account} += $how_much;
        }
    }
    print "\n";
}
```

The following example, which is considerably shorter, uses a hash to find out if a list of strings contains only unique strings:

```
while($string = <>)
{
    chomp($string);
    if (exists($myhash{$string}))
    {
        print "The string \"" . $string . "\" was already encountered!";
        last;
    }
    else
    {
        $myhash{$string} = 1;
    }
}
```

## 2.1. Hash-Related Functions ¶

### delete

delete can be used to remove a key or a set of keys out of a hash. For example:

```
$myhash{"hello"} = "world";
$myhash{"perl"} = "TMTOWTDI";
$myhash{"shlomi"} = "fish";

if (exists($myhash{"perl"}))
{
    print "The key perl exists!\n";
}
else
{
    print "The key perl does not exist!\n";
}

delete($myhash{"perl"});

if (exists($myhash{"perl"}))
{
    print "The key perl exists!\n";
}
else
{
    print "The key perl does not exist!\n";
}
```

### The Comma Regarding Hashes

The comma can be used to combine two arrays into one larger one. Given the fact that a mini-hash with one key and one value can be specified using the `$key => $value` notation (which is essentially equivalent to `$key, $value`) a hash can be initialized in one statement.

Here's an example:

```
%hash1 = (
    "shlomi" => "fish",
    "orr" => "dunkelman",
    "guy" => "keren"
    );

%hash2 = (
    "george" => "washington",
    "jules" => "verne",
    "isaac" => "newton"
    );

%combined = (%hash1, %hash2);

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

If the two combined hashes contain several identical keys, then the values of the latter hash will win.

## 3. Declaring Local Variables with "my" ¶

Variables can be made local to their own scope by declaring them using a keyword called `my`. When the interpreter leaves the scope, the variable will be restored to its original value.

Here's an example:

```
$x = 5;
$y = 1000;
{
    my ($y);
    for($y=0;$y<10;$y++)
    {
        print $x, "*", $y, " = ", ($x*$y), "\n";
    }
}

print "Now, \$y is ", $y, "\n";
```

If you wish to declare more than one variable as local you should use a set of parenthesis surrounding the variable list. For example: `my (@array, $scalar1, %hash, $scalar2);`. If you want to declare only one variable, then they are optional.

## 3.1. "use strict", Luke! ¶

Perl has a pragma (= an interpreter-related directive) known as `use strict;`, which among other things, makes sure all the variables you use will be declared with `my`. If you reference a variable that was not declared with `my` it will generate an error.

Using `use strict` is in most cases a good idea, because it minimises the number of errors because of typos. Just type `use strict;` at the beginning of your program and you can sleep better at nights.

As an example, here is the primes program, `use strict`-ed:

```
use strict;
use warnings;

my (@primes, $i);

MAIN_LOOP: for(
    @primes=(2), $i=3 ;
    scalar(@primes) < 200 ;
    $i++
    )
{
    foreach my $p (@primes)
    {
        if ($i % $p == 0)
        {
            next MAIN_LOOP;
        }
    }

    push @primes, $i;
}

print join(", " , @primes), "\n";
```

Notice the use of `my` in the declaration of the `foreach` loop. Such just-in-time declarations, inspired by C++, are possible in perl.

## 3.2. "use warnings", Luke! ¶

There is another pragma which is called and typed as `use warnings;`, that causes the interpreter to emit many warnings to the command window in case you're doing wrong things (like using `undef` in an expression). It is also very useful to declare it at the beginning of the program, in addition to `use strict;`. So, from now on, our programs will include it as well.

## 4. Functions ¶

We already encountered some of Perl's built-in functions. Perl enables us to define our own functions using Perl code. Whenever you use a piece of code more than once in a program, it is a good idea to make it into a function. That way, you won't have to change it in more than one place.

In Perl, every function accepts an array of arguments and returns an array of return values. The arguments (also known as "parameters") can be found in the `@_` variable. This variable is magical and need not and should not be declared with `my`. In order to return values from a function, one can use the `return` keyword.

To declare a function use type `sub function_name {` at the beginning and a `}` at the end. Everything in between, is the function body.

Here's an example, for a function that returns the maximum and the minimum of an array of numbers.

```
use strict;
use warnings;

sub min_and_max
{
    my (@numbers);

    @numbers = @_;

    my ($min, $max);

    $min = $numbers[0];
    $max = $numbers[0];

    foreach my $i (@numbers)
    {
        if ($i > $max)
        {
            $max = $i;
        }
        elsif ($i < $min)
        {
            $min = $i;
        }
    }

    return ($min, $max);
}

my (@test_array, @ret);
@test_array = (123, 23 , -6, 7 , 80, 300, 45, 2, 9, 1000, -90, 3);
```

```perl
@ret = min_and_max(@test_array);

print "The minimum is ", $ret[0], "\n";
print "The maximum is ", $ret[1], "\n";
```

And here's another one for a function that calculates the hypotenuse of a right triangle according to its other sides:

```perl
use strict;
use warnings;

sub pythagoras
{
    my ($x, $y);

    ($x, $y) = @_;

    return sqrt($x**2+$y**2);
}

print "The hypotenuse of a right triangle with sides 3 and 4 is ",
    pythagoras(3,4), "\n";
```

## 4.1. Function Recursion ¶

Functions can call other functions. In fact, a function can call itself, either directly or indirectly. When a function calls itself it is known as **recursion**, a Computer Science methodology which can be implemented with or without functions.

Here's an example where recursion is used to find all the permutations of splitting 10 among three numbers:

```perl
use strict;
use warnings;

sub mysplit
{
    my ($total, $num_elems, @accum) = @_;

    if ($num_elems == 1)
    {
        push @accum, $total;
        print join(",", @accum), "\n";

        return;
    }
    for (my $item=0 ; $item <= $total ; $item++)
    {
        my @new_accum = (@accum, $item);
        mysplit($total-$item, $num_elems-1, @new_accum);
    }
}

mysplit(10,3);
```

A couple of notes are in place. First of all, perl does not handle tail recursion very well, at least not in the current incarnation of the compiler. If your recursion can be done using a simple loop, do it with it.

Secondly, some systems (such as Microsoft Windows) limit an executable to a certain amount of stack, as far as such languages as Assembler or C are concerned. This should not be of any concern to perl hackers, because the perl interpreter does not translate a perl function call into a C function call. (not to mention that the perl interpreter on those systems is compiled with enough stack for itself).

Sometimes, recursion is helpful, but if you see that your recursion is getting too deep, you should consider using your own dedicated stack (which can be implemented as an array) instead. It's a good programming practice.

## 4.2. Use of "shift" in Functions ¶

One can use the shift function to extract arguments out of the argument array. Since this use is so common, then simply typing shift without any arguments will do exactly that.

Here is the split program from the previous slide, which was re-written using shift:

```
use strict;
use warnings;

sub mysplit
{
    my $total = shift;
    my $num_elems = shift;
    my @accum = @_;

    if ($num_elems == 1)
    {
        push @accum, $total;
        print join(",", @accum), "\n";

        return;
    }

    for (my $item = 0 ; $item <= $total ; $item++)
    {
        my @new_accum = (@accum, $item);
        mysplit($total-$item, $num_elems-1, @new_accum);
    }
}

mysplit(10,3);
```

## 5. File Input/Output ¶

By now you are probably wondering how perl can be used to interact with the external world, and this is where **File Input/Output** enters the frame.

In Perl, file I/O is handled by using sessions: you are opening a file for reading or writing (or both), do with it what you want, and then close it. In Perl, filehandles implemented as the so-called globs are placed on a separate namespace than that of the variables. It is generally marked with a starting asterisk (*), which can be omitted if the first letter is a capital one.

To open a file use the `open my $my_file_handle, $mode, $file_path;` notation, and to close a file use the `close($my_file_handle);` notation. The `$mode` determines whether the file will be open for reading, writing, appending or some of them. The following table should give you a quick reference:

| | |
|---|---|
| > | Writing (the original file will be erased before the function starts). |
| < (or nothing) | Reading |
| >> | Appending (the file pointer will start at the end and the file will not be overridden) |
| +< | Read-write, or just write without truncating. |

`$file_path` is the pathname of the file to open relative to the script current working directory (**CWD**). For instance, the command `open I, "<", "../hello.txt";` opens the file "hello.txt" found a directory above the current directory for reading.

### 5.1. Using "print" with Files ¶

The print command which we encountered before can also be used to send output to files. In fact, the screen itself is a special filehandle, whose name is `STDOUT`, but since its use is so common, perl allows it to be omitted.

The syntax of printing to a file is `print File $string1, $string2, ...`. Here's a short example that prepares a file with a pyramid in it:

```


#!/usr/bin/env perl

use strict;
use warnings;

my $pyramid_side = 20;
```

```
open my $out, ">", "pyramid.txt";
for ($l=1 ; $l <= $pyramid_side ; $l++)
{
    print {$out} "X" x $l;
    print {$out} "\n";
}
close($out);
```

In order to print to more than one file at once, one needs to use two separate print statements. Here's an example, that prints to one file the sequence 0, 0.1, 1, 1.1, 2, 2.1, 3, 3.1⋯, and to the other the sequence 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5⋯.

```
use strict;
use warnings;

open my $seq1, ">", "seq1.txt";
open my $seq2, ">", "seq2.txt";

for (my $x=0 ; $x < 100 ; $x++)
{
    print {$seq1} $x, "\n";
    print {$seq2} $x, "\n";
    print {$seq1} ($x+0.1);
    print {$seq2} ($x+0.5);
    print {$seq1} "\n";
    print {$seq2} "\n";
}

close($seq1);
close($seq2);
```

## 5.2. The <FILEHANDLE> Operator ¶

Just like print can be generalised to files, so can the <> which we encountered before. If you place the name of the filehandle inside the operator, it will read a line from the file opened by that filehandle.

Here's an example, let's append the line numbers to a given file:

```
use strict;
use warnings;

my ($line_num, $line);

$line_num = 0;
open my $in, "<", "input.txt";
open my $out, ">", "output.txt";

while ($line = <$in>)
{
    # We aren't chomping it so we won't lose the newline.
    print {$out} $line_num, ": ", $line;
    $line_num++;
}

close ($in);
close ($out);
```

And the following example counts the number of lines in a file that start with the letter "A" (case-insensitive).

```
use strict;
use warnings;

my ($filename, $lines_num, $line, $c);

$lines_num = 0;
$filename = "input.txt";
open my $in,  "<", $filename;
while ($line = <$in>)
{
    $c = substr($line, 0, 1);
    if (lc($c) eq "a")
    {
        $lines_num++;
    }
}
close($in);

print "In " , $filename, " there are ",
    $lines_num, " lines that start with \"A\".\n";
```

The `join("", <FILEHANDLE>)` command returns the entire contents of the file from the current position onwards, and may prove to be useful. Examples for it will be given in the next section where regular expressions will be taught.

## 6. The @ARGV Array ¶

One can use the `@ARGV` variable to access the command line arguments passed to the perl script at the command line. Here's an example, that makes a backup of the file specified as its argument:

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

```
my $filename = $ARGV[0];

open my $in, "<", $filename;
open my $out, ">", $filename.".bak";
print {$out} join("",<$in>);
close($in);
close($out);
```

Using the command-line for specifying parameters to the program is usually more handy than using files, or modifying the script each time.

Note that it is often convenient to use `shift` to draw arguments out of `@ARGV` one at a time. When used without parameters outside of functions, shift extract arguments out of `@ARGV`.

## 7. Regular Expressions ¶

Regular expressions are one of the most powerful features of Perl, but are also one of those that beginners find most difficult to understand. Don't let that fact discourage you: they are definitely worth your time.

Basically, a regular expression tells the interpreter to search a string for a pattern. The pattern can be a simple one like the word "hello" or a complex one, such as the word "hello" followed by any number of instances of the word "one", followed by the word "triple". The latter will be written as `hello(?:one)*triple` inside a regular expression.

The perl operators allow you to do three things with a regular expression:

1. Check if it exists in the string at all.
2. Retrieve various captures out of it.
3. Retrieve all of its occurrences.
4. Substitute the first occurrence or all of its occurrences with some other string or (perl) expression.

Some perl functions such as `split` and `grep` (which will be covered later) can utilise regular expressions to do other tasks.

### 7.1. Syntax ¶

The syntax for checking if a string contains the regular expression is `$string =~ /$regexp/`. For instance the following program checks if a given string contains the word hello (in all lowercase):

```
use strict;
use warnings;

my $string = shift(@ARGV);

if ($string =~ /hello/)
```

```
{
    print "The string contains the word \"hello\".\n";
}
else
{
    print "The string does not contain the word \"hello\".\n";
}
```

Note that alphanumeric characters in regular expressions stand for themselves, and that behaviour is guaranteed not to change in further versions of perl. Other characters may need to be prefixed with a backslash (`\`) if placed inside a regular expression.

## 7.2. ".", "[ ... ]" ¶

In this slide we will learn how to specify any character or that a character will be one of a range of several possible characters.

### The "." stands for any character

By putting a `.` character inside a regular expression, it means that it can match any character, excluding a newline. For example, the following snippet matches 5 letter words that start with 'l' and end with 'x':

```
use strict;
use warnings;

my $string = lc(shift(@ARGV));

if ($string =~ /l...x/)
{
    print "True\n";
}
else
{
    print "False\n";
}
```

### The [ ... ] specifies more than one option for a character

When square brackets appear, one can specify more than one character inside them as option for matching. If the first character is `^` then they will match everything that is **not** one of the characters.

One can specify a range of characters with the hyphen. For example the pattern `[a-zA-Z0-9_]` matches every alpha-numeric character.

Here's an example that checks if a valid identifier for a perl variable is present in the string:

```
use strict;
use warnings;

my $string = lc(shift(@ARGV));

if ($string =~ /\$[A-Za-z_]/)
{
    print "True\n";
}
else
{
    print "False\n";
}
```

## 7.3. The "*" and Friends ¶

In order to repeat a character more than one time, several repetition patterns can be used. The asterisk (`*`) can be used to indicate that the character should repeat zero or more times. The plus sign (`+`) can be used to indicate that the character should repeat one or more times. As for the question mark (`?`) it indicates an appearance of zero or one time.

The following pattern `/ab*c/` matches "a", followed by as many times of b as we like (including zero), and then c. The pattern `/you k?now/` matches both "you know" and "you now". The pattern `/hello +world/` matches the word "hello" followed by the word "world". (with some whitespace in between).

## 7.4. Grouping ¶

One can use the cluster grouping notation (`(?: ... )`) or the capture grouping notation (`( ... )`) to group several characters so the entire group can be repeated with `+`, `*` or `?`.

The difference between clustering and capturing, is that with capturing the Perl interpreter also keeps track of the text matched by the captures in the `$1`, `$2`, `$3`, etc. variables. Clustering as a result is faster and less intrusive.

For example, the following perl program accepts a string as an argument, and checks if it is an entire sentence that starts with "the" and ends with "there":

```
use strict;
use warnings;

my $string = lc(shift(@ARGV));

if ($string =~ /the(?: +[a-z]+)* +there/)
{
    print "True\n";
}
else
{
    print "False\n";
}
```

It is possible to nest groupings, so for example the following matches a square brackets-enclosed semicolon separated list of curly braces-enclosed comma-separated lists of numbers:

```
use strict;
use warnings;

my $string = lc(shift(@ARGV));

if ($string =~ /\[\{[0-9]+(?:,[0-9]+)+\}(?:;\{(?:[0-9]+(?:,[0-9]+)+)\})+\]/)
{
    print "True\n";
}
else
{
    print "False\n";
}
```

So it matches strings like `[{54,129};{236,78}]` and `[{54,129};{236,78};{78,100,808};{72,1009,8}]`

## 7.5. Alternation with "|" ¶

One can specify several possible matching for an expression to match with `|`. For instance, `dog|cat|mouse` will match either "dog" or "cat" or "mouse". But naturally the options can themselves be regular expression which are as complex as perl supports.

The alternation operator will try to match as much of the expression as it can, so it is recommended to use grouping to bound it. (e.g: `(?:dog|cat|mouse)`)

The following pattern matches a sequence of whitespace-delimited "words" that each may contain either a set of letters or a set of digits (= a positive integer):

```
(?:[a-zA-Z]+|[0-9]+)(?: +(?:[a-zA-Z]+|[0-9]+))*
```

## 7.6. Binding to the Beginning or the End of a String ¶

In a regular expression a caret (`^`) at the beginning of the expression matches the beginning of the string and a dollar sign (`$`) at the **end** of the expression matches the end of the string.

For example, the following program checks if a string is composed entirely of the letter A:

```
use strict;
use warnings;

my $string = shift;

if ($string =~ /^[Aa]*$/)
{
    print "The string you entered is all A's!\n";
}
else
{
    print "The string you entered is not all A's.\n";
}
```

It's much shorter than with using a loop and also much faster.

Here's another example. Let's check if a string ends with three periods:

```
use strict;
use warnings;

my $string = shift;

if ($string =~ /\.\.\. *$/)
{
    print "The string you entered ends with an ellipsis.\n";
}
else
{
    print "The string you entered does not end with an ellipsis.\n";
}
```

## 7.7. Substituting using Regular Expressions ¶

One can use regular expression to substitute an occurrence of a regular expression found inside a string to something else. The operator used for this is `s/$exp_to_search/$exp_to_subs_for/`. The following simple program replaces a number with the string "Some Number":

```
use strict;
use warnings;

my $string = shift;

$string =~ s/[0-9]+/Some Number/;
```

```
print $string, "\n";
```

Notice that in this case, only the first occurrence of the string will be substituted. The g switch which will be covered soon, enables us to substitute **all** of the occurrences.

One can include the captures that are matched in the expression as part of the substitution string. The first capture is $1, the second $2 etc.

Here's an example which reverses the order of two words at the beginning of a sentence:

```
use strict;
use warnings;

my $string = shift;

$string =~ s/^([A-Za-z]+) *([A-Za-z]+)/$2 $1/;

print $string, "\n";
```

**Note**

The $1, $2 variables are also available after the regular expression match or substitution is performed, and can be used inside your perl code.

### 7.7.1. The "e" Switch ¶

If an e is appended to the end of the substitution command, then the right side is treated as a normal Perl expression, giving you the ability to use operators and functions.

The following example, replaces the first word in the sentence with its length:

```
use strict;
use warnings;

my $string = shift;

$string =~ s/^([A-Za-z]+)/length($1)/e;

print $string, "\n";
```

And the following example, converts the first word that starts with the letter "S" into uppercase:

```
use strict;
use warnings;

my $string = shift;

$string =~ s/([Ss][A-Za-z]*)/'<'.uc($1).'>'/e;

print $string, "\n";
```

### 7.7.2. Ungreedy Matching with *? and Friends ¶

By default, the regular expression repetition operators such as `*`, `+` and friends are greedy. Hence, they will try to match as much as possible of the string as they can.

If you wish them to match as little as possible, append a question mark (`?`) after them. Here's an example:

```
use strict;
use warnings;

my $string = "<html>Hello</html>You</html>";

my $greedy = $string;
$greedy =~ s/<html>.*<\/html>/REPLACED/;

my $ungreedy = $string;
$ungreedy =~ s/<html>.*?<\/html>/REPLACED/;

print $string, "\n",
    $greedy, "\n",
    $ungreedy, "\n";
```

## 7.8. Useful Flags ¶

There are other flags than `e` that can be appended to the end of the match or substitution.

An `i` following the regular expression call, causes a case-insensitive operation on the string. Thus, for example, an "A" will match both "a" and "A". Note that the strings placed in `$1`, `$2` and friends will still retain their original case.

A `g` causes the match or substitution to match **all** occurrences, not just one. If used with a match in an array context (e.g: `@occurrences = ($string =~ /$regexp/g);`) it retrieves all the matches and if used with a substitution it substitutes all the occurrences with the string.

This example replaces all the occurrences of the word "hello" by the index of their occurrence:

```
use strict;
use warnings;

my $index = 0;
sub put_index
{
    $index++;
    return $index;
}

my $string = shift;

$string =~ s/hello/put_index()/gei;

print $string, "\n";
```

## 7.9. Useful Escape Sequences ¶

Perl has several escape sequences which are specific to regular expressions. Here is a list of some of the more useful ones:

- `\w` - matches a word character, equivalent to `[A-Za-z_]`. The corresponding sequence - `\W` matches a non-word character.
- `\s` - matches a whitespace character, usually space and tab. `\S` matches a non-whitespace character.
- `\d` - matches a digit. Equivalent to `[0-9]`. `\D` matches a non-digit.
- `\b` - matches a word boundary. This sequence in fact has a zero width but it is nevertheless highly useful. `\B` matches something that is not a word boundary.

## 7.10. The Next Step ¶

This was just an introduction to perl's regular expressions. There is much more available functionality, and I am referring you to the following sources for more information:

- The perlsyn document.
- perlrequick (the regular expressions' quick start), perlretut (regular expressions tutorial) and perlreref (regular expressions reference).
- The perlre document. The bugs section of it says that it "varies from difficult to understand to completely and utterly opaque".
- The perlfaq6 document that deals with regular expressions.
- The book *Mastering Regular Expressions* by Jeffrey Friedl, published by O'Reilly and Associates.

## 8. String Interpolation ¶

Perl supports inserting variables into string constants simply by placing their name along with the dollars inside them. Here's an example:

```
use strict;
use warnings;

my $name;
print "Please enter your name:\n";
$name = <>;
chomp($name);
print "Hello, $name!\n";
```

Note that once perl encounters a `$`-sign, it will try to use as many characters as possible from the string as the variable name, even if a variable by that name does not exist. Therefore, if you write `$xy` inside a string, it will not take the value of `$x` and append the string "y" to it! To overcome this limitation, you can limit the variable name using curly braces: `"Hello ${boy}and${girl}"`.

In any case, interpolation is especially useful for building regular expressions, since the string may contain control characters.

## 9. Useful Functions ¶

Perl has many functions that allow batch operations on entire arrays. Those functions are not only convenient, but are also faster than unrolling them into `for` and `foreach` loops.

This behaviour is sometimes referred to as the **perl pipeline** because it resembles the pipeline used by the UNIX shells.

### 9.1. split ¶

The `split` function can be used to split a string according to a regular expression. The syntax of the split function is as follows:

`@components = split(/$regexp/, $string);`

Here's a simple example that retrieves the user-id of a given username:

```
use strict;
use warnings;

my ($line, @parts);
my $user_name = shift;

open my $in, "<", "/etc/passwd";
while ($line = <$in>)
{
    @parts = split(/:/, $line);
    if ($parts[0] eq $user_name)
    {
        print $user_name . "'s user ID is " . $parts[2] . "\n";
        exit(0);
    }
}
close($in);

print $user_name . "'s user ID was not found!\n";
exit(-1);
```

The following code splits a sentence into words and prints them:

```
use strict;
use warnings;

my $sentence = shift;

my @words = split(/\s+/, $sentence);

my $i;
for($i=0;$i<scalar(@words);$i++)
{
    print "$i: " . $words[$i] . "\n";
}
```

## 9.2. map ¶

The map function traverses an array and maps each element to one or more ( or zero) elements in a new array. It has a rather unorthodox syntax in that it is written as `@new_array = (map { <Some Expression with $_> } @array)` .

For each element of the array @array, the variable `$_` is assigned its value, and within the curly brackets one can put an expression that is dependant on it.

The following example multiplies each element of an array by two:

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

```
my @array = (20, 3, 1, 9, 100, 88, 75);

my @new_array = (map { $_*2; } @array);

print join(",", @new_array), "\n";
```

Using map is usually faster than using a `foreach $elem (@array) { ... push @new_array, $new_elem; }` loop, at least when the function performed is a relatively uncomplex one.

The following program decodes a run-length encoding compression, in which each element of the array is a number followed by its number of times:

```
use strict;
use warnings;

my $default_input_string = "2-5,3-9,1-2,8-1,4-7,5-9,20-3,16-9";

my $input_string = shift || $default_input_string;

# RLE stands for Run-Length Encoding
my @rle_components = split(/,/, $input_string);
my @expanded_sequence = (map
    {
        my ($what, $times) = split(/-/, $_);
        (($what) x $times);
    }
    @rle_components
    );

print join(",", @expanded_sequence), "\n";
```

As you can see, the expression at the end of the map iterator can be an array with more than one element. It can also be an empty array, which means that some elements can be filtered out.

## 9.3. sort ¶

The `sort` function can sort an array based on a comparison expression. As with the `map` function, this expression can be as complex as you'd like and may actually include a call to a dedicated function.

Within the comparison block, the variables `$a` and `$b` indicate the two elements to be compared. If the expression returns a negative value it means that `$a` should precede `$b`. If it is positive, it means that `$b` should come before `$a`. If it is zero, it indicates that it does not matter which one will come first.

The following example sorts an array of integers numerically:

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

```
my @array = (100,5,8,92,-7,34,29,58,8,10,24);

my @sorted_array =
    (sort
        {
            if ($a < $b)
            {
                return -1;
            }
            elsif ($a > $b)
            {
                return 1;
            }
            else
            {
                return 0;
            }
        }
        @array
    );

print join(",", @sorted_array), "\n";
```

### 9.3.1. <=> and cmp ¶

Perl has two operators `<=>` and `cmp`, which are very useful when wishing to sort arrays. `$x <=> $y` returns -1 if `$x` is numerically lesser than `$y`, 1 if it's greater, and zero if they are equal.

`cmp` does the same for string comparison. For instance the previous example could be re-written as:

```
use strict;
use warnings;

my @array = (100,5,8,92,-7,34,29,58,8,10,24);

my @sorted_array = (sort { $a <=> $b } @array);

print join(",", @sorted_array), "\n";
```

Much more civil, isn't it? The following example, sorts an array of strings in reverse:

```
use strict;
use warnings;

my @input = (
    "Hello World!",
    "You is all I need.",
    "To be or not to be",
    "There's more than one way to do it.",
    "Absolutely Fabulous",
    "Ci vis pacem, para belum",
    "Give me liberty or give me death.",
    "Linux - Because software problems should not cost money",
);

# Do a case-insensitive sort
my @sorted = (sort { lc($b) cmp lc($a); } @input);

print join("\n", @sorted), "\n";
```

## 9.4. grep ¶

The grep function can be used to filter items out of an array based on a boolean expression or a regular expression. The syntax for the block usage is similar to map while the syntax for the regexp usage is similar to split.

Here is an example that takes a file and filters only the perl comments whose length is lesser than 80 characters:

```


use strict;
use warnings;

my (@lines, $l);

my $filename = shift;

open my $in, "<", $filename;
while ($l = <$in>)
{
    chomp($l);
    push @lines, $l;
}
close($in);

# Filter the comments
my @comments = grep(/^#/, @lines);
# Filter out the long comments
my @short_comments = (grep { length($_) <= 80 ; } @comments);

print join("\n", @short_comments), "\n";
```

And here's how grep can help us find the first 100 primes:

```perl
use strict;
use warnings;

my @primes = (2);
for(my $n=3 ; scalar(@primes) < 100 ; $n++)
{
    if (scalar(grep { $n % $_ == 0 ; } @primes) == 0)
    {
        push @primes, $n;
    }
}
print join(", ", @primes), "\n";
```

## 10. References ¶

Perl allows one to refer to the location of a certain variable in memory. An expression that holds such a location is called a **reference**. Those that are familiar with C's or Pascal's pointers may think of references as pointers. There are however, two fundamental differences:

1. There is no reference arithmetics in perl. If for example, a reference points to the fifth element of an array, then adding 1 to it will not refer you to the sixth element. In fact, adding or subtracting integers from references is possible but quite meaningless.
2. A reference is guaranteed to remain the same even if an array or a string are resized. In C, `realloc`ing an array yields a completely different pointer.

Perl distinguishes between an array or a hash and a reference of it. The reference of any array may be taken, and a reference to an array may always be converted to its elements, but there is still a difference in functionality.

The best way to change a variable in a different scope (such as inside a different function) is to pass its reference to the function. The called function can then dereference the variable to access or modify its value.

### 10.1. Example: The Towers of Hanoi ¶

In this example, which is intended to give a taste of the capabilities of references, we will solve the well-known Towers of Hanoi problem. (Refer to the link to learn more about the problem.) The number of disks can be input from the command-line. The towers themselves will be represented as an array of three elements, each of which is a reference to an array.

We will use the recursive solution in which in order to move a column of $N disks, we first move the upper column with $N-1 disks and then move the bottom most disk, and then move the $N-1 disks-long column on top of it.

Here goes:

```
use strict;
use warnings;

my $num_disks = shift || 9;

my @towers = (
    [ reverse(1 .. $num_disks) ],  # A [ ... ] is a dynamic reference to
    [ ],                           # an array
    [ ]
    );

sub print_towers
{
    for(my $i=0 ; $i < 3 ; $i++)
    {
        print ": ";
        print join(" ", @{$towers[$i]}); # We de-reference the tower
        print "\n";
    }
    print "\n\n";
}

sub move_column
{
    my $source = shift;
    my $dest = shift;
    my $how_many = shift;

    if ($how_many == 0)
    {
        return;
    }
    # Find the third column
    my $intermediate = 0+1+2-$source-$dest;
    move_column($source, $intermediate, $how_many-1);
    # Print the current state
    print_towers();
    # Move one disk. Notice the dereferencing of the arrays
    # using @{$ref}.
    push @{$towers[$dest]}, pop(@{$towers[$source]});
    move_column($intermediate, $dest, $how_many-1);
}

# Move the entire column
move_column(0,1,$num_disks);
print_towers();
```

## 10.2. \ - Taking a Reference to an Existing Variable ¶

In Perl the backslash (\) is an operator that returns the reference to an existing variable. It can also return a dynamic reference to a constant.

Here is an example that uses a reference to update a sum:

```
use strict;
use warnings;

my $sum = 0;

sub update_sum
{
    my $ref_to_sum = shift;
    foreach my $item (@_)
    {
        # The ${ ... } dereferences the variable
        ${$ref_to_sum} += $item;
    }
}

update_sum(\$sum, 5, 4, 9, 10, 11);
update_sum(\$sum, 100, 80, 7, 24);

print "\$sum is now ", $sum, "\n";
```

As can be seen, because the reference to $sum was used, its value changes throughout the program.

## 10.3. [ @array ] - a Dynamic Reference to an Array ¶

An array surrounded by square brackets ([ @array ]) returns a dynamic reference to an array. This reference does not affect other values directly, which is why it is called dynamic.

We have already encountered such dynamic array references in the Hanoi example. But their use is not limited to what we've seen there. Since a reference to an array is a scalar, it can serve as a hash value and therefore serve as an object member. (as will be seen later in the series).

In this example, a function is given two arrays, and returns an array that is the element-wise sum of both of them:

```
use strict;
use warnings;

sub vector_sum
{
    my $v1_ref = shift;
    my $v2_ref = shift;

    my @ret;
    my @v1 = @{$v1_ref};
    my @v2 = @{$v2_ref};

    if (scalar(@v1) != scalar(@v2))
    {
        return undef;
    }
    for(my $i=0;$i<scalar(@v1);$i++)
    {
        push @ret, ($v1[$i] + $v2[$i]);
    }

    return [ @ret ];
}

my $ret = vector_sum(
    [ 5, 9, 24, 30 ],
    [ 8, 2, 10, 20 ]
);

print join(", ", @{$ret}), "\n";
```

### Note

The fundamental difference between using `\@myarray` on an existing variable named `@myarray` to using `[ @myarray ]` is this: the latter form creates a dynamic copy of `@myarray` and if this copy changes, `@myarray` will not change with it. On the other hand, changes made to a reference generated by backslash, will affect the original variable.

Note that if the members of `@myarray` are themselves references, then the second form will not make a deep copy of them. Thus, they can be modified too, even in the second form.

## 10.4. { %hash } - a Dynamic Reference to a Hash ¶

In a similar way to the square brackets, putting a hash inside a pair of curly braces (`{ ... }`) will make it into a reference to a hash. Like an array reference, this reference is a scalar and can be used as an array element or a hash value. Plus, its own values can be references to other arrays or hashes.

To demonstrate this let's see the code of part of the contents table of this very lecture, to demonstrate the multi-level data structure capabilities of perl:

```perl
use strict;
use warnings;

my $contents =
{
    'title' => "Contents",
    'subs' =>
    [
        {
            'url' => "for",
            'title' => "The for loop",
            'subs' =>
            [
                {
                    'url' => "next.html",
                    'title' => "Behaviour of next in the for loop",
                },
                {
                    'url' => "whence_for.html",
                    'title' => "Whence for?",
                },
            ],
        },
        {
            'url' => "hashes",
            'title' => "Hashes",
            'subs' =>
            [
                {
                    'url' => "functions.html",
                    'title' => "Hash Functions",
                },
            ],
        },
        {
            'url' => "my",
            'title' => "Declaring Local Variables with \"my\"",
            'subs' =>
            [
                {
                    'url' => "use_strict.html",
                    'title' => "\"use strict\", Luke!",
                },
            ],
        },
        {
            'url' => "argv.html",
            'title' => "The \@ARGV array",
        },
    ],
    'images' =>
    [
        {
            'url' => 'style.css',
            'type' => 'text/css',
        }
    ],
};

sub get_contents
{
    return $contents;
}
```

## 10.5. The Arrow Operators ¶

An arrow (`->`) followed by a square or curly brackets can be used to directly access the elements of an array or a hash referenced by a certain hash. For instance: `$array_ref->[5]` will retrieve the 5th element of the array pointed to by `$array_ref`.

As an example let's print a tree of the contents of the part of the lecture that was presented in the previous slide:

```
use strict;
use warnings;

do "lol.pl";         # Load the other file

my $cont = get_contents();

print $cont->{'title'}, "\n";
print $cont->{'subs'}->[0]->{'url'}, "\n";
print $cont->{'subs'}->[0]->{'subs'}->[1]->{'title'}, "\n";
```

Note that the arrows following the first arrow are optional as perl sees that the programmer wishes to access the subseqeunt sub-items. However, the first one is mandatory because the expression `$array_ref{'elem'}` looks for the hash `%array_ref`.

## 10.6. Dereferencing ¶

The entire scalar or data structure pointed to by the reference can be retrieved by dereferencing. Dereferencing is done by using a `$`, a `@` or a `%` (depending if the reference refers to a scalar , array or a hash respectively), and then the reference inside curly braces.

Here are some simple examples:

```perl
use strict;
use warnings;

my $ds1 =
{
    'h' => [5,6,7],
    'y' => { 't' => 'u', 'o' => 'p' },
    'hello' => 'up',
};

my $array_ref = [5, 6, 7, 10, 24, 90, 14];
my $x = "Hello World!";
my $y = \$x;

print "\$array_ref:\n";

print join(", ", @{$array_ref}), "\n";


print "\n\n\$ds1->{'h'}:\n";

print join(", ", @{$ds1->{'h'}}), "\n";

my %hash = %{$ds1->{'y'}};

print "\n\n\%hash:\n";

foreach my $k (keys(%hash))
{
    print $k,  " => ", $hash{$k};
}


print "\n\n\$\$y:\n";

print ${$y}, "\n";
```

If the expression that yields the reference is a simple one than the curly brackets can be omitted (e.g: `@$array_ref` or `$$ref`). However, assuming you use curly brackets - the expression surrounded inside them can be as complex as you would like.

## 11. Using the perl Debugger ¶

Even if your perl code compiles and runs, it doesn't mean it is error free. Very often you will write programs that when given specific input, will behave very differently from their intended behaviour. Perl supplies the user with a debugger that can help in trapping such bugs in the program.

The debugger is invoked simply by appending a "-d" switch before the script name. E.g: perl -d hello.pl. After typing "Return" at the shell command line, you will have an interactive debugging session at your disposal. At this session, you can control the execution of your program as well as enter any Perl command you like.

The following chapter will cover the basics of using the perl debugger, and will give you pointers on where to find more information.

## 11.1. Stepping over and Stepping in ¶

The debugger displays the perl statement that is about to be executed in its display window, and below it the command prompt. It looks something like this:

```
shlomi:~/bin# perl -d add_cr.pl
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main::(add_cr.pl:5):    my ($contents);
  DB<1>
```

The most commonly used command in the debugger is to have it execute one (and only one) perl statement. This is called stepping and there are two types of it: stepping over and stepping in.

The difference is that stepping over does not enter the perl functions that were called in the expression that was evaluated. Stepping in, on the other hand, does enter such functions.

To step over type n (short for "next") followed by enter , and you'll see the next perl command displayed. To step in type s (short for "step").

And here's how the screen will look after a few step overs:

```
shlomi:~/bin# perl -d add_cr.pl test.txt
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main::(add_cr.pl:5):    my ($contents);
  DB<1> n
main::(add_cr.pl:7):    foreach my $file (@ARGV)
main::(add_cr.pl:8):    {
  DB<1> n
```

```
main::(add_cr.pl:9):        $contents = "";
  DB<1> n
main::(add_cr.pl:11):       open my $in, "<", $file;
  DB<1>
```

Note that sometimes an expression will take more than one step over to pass. Operations such as `map`, `sort` and friends are especially notorious for that. Breakpoints that will be covered in the next slide offer a solution to this problem.

## 11.2. Setting Breakpoints and Continuing ¶

When the user places a breakpoint on a certain line in a program, they instruct the debugger to stop the execution of the program when this line is reached. Then, one can cause the program to run freely using the continue command, and have it stop only at this line.

Breakpoints generally save a lot of stepping. To set a breakpoint type `b $line_number` or `b $function_name` where $line_number is the line number and $function_name is the function name. A breakpoint on a function will stop the execution of the program as soon as the function is entered.

To continue the execution of the program until a breakpoint is reached (or until the program terminates) type `c`. Here is an example session with the same program:

```
shlomi:~/bin# perl -d add_cr.pl test.txt
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main::(add_cr.pl:5):    my ($contents);
  DB<1> b 11
  DB<2> c
main::(add_cr.pl:11):       open my $in, "<", $file;
  DB<2>
```

One can set a conditional breakpoint by typing `b [line] [perl expression]`. A conditional expression is one that stops the execution of the program only if the perl expression evaluates to true. The perl expression can be as complex as you like and may include whitespace.

To delete a breakpoint type `d [line]`. After a breakpoint is deleted it will no longer affect the execution of your program.

## 11.3. Executing Perl Commands inside the Debugger ¶

One can execute perl commands inside the debugger. For example, typing `print $x` at the debugger prompt will print the value of the `$x` variable. You can also modify the values of variables in this way, or run functions, etc.

One special command that is also useful is `x`. `x $var` displays `$var` in a hierarchical manner, which is very useful for lists of lists etc. The **Data::Dumper** module which is available from CPAN offers a similar functionality, from within your perl programs.

### 11.4. Getting More Information ¶

The `h` command in the debugger displays the list of available debugger commands.

Usage information about the perl debugger is available at the perldebug document.

There are some front-ends to the perl debugger. One of them, which runs on Linux, is ddd.

## 12. Finale ¶

*I'm reminded of the day my daughter came in, looked over my shoulder at some Perl 4 code, and said, "What is that, swearing?"*

*(Larry Wall (the father of Perl) in <199806181642.JAA10629@wall.org>)*

But luckily the next back-end for Perl 5 - Parrot will support multiple syntax front-ends, so one will be able to write Perl in an alternate syntax that does not have all those pesky sigils. The question is: why would anyone ever want to do that? **:-)**

No, seriously. We hoped you enjoyed the lecture series, but remember that it does not teach you everything that is to know about perl, much less how to understand other people's code.

By now, you can probably understand the perl core documents, so it is recommended that you read them in case you are looking for more information. You can also refer to the Perl Beginners' Site for more sources of information as well as ways to get help online.

We hope to continue this lecture with more presentations covering Perl's modules, objects and references to functions. Until then, stay cool and may the dollar sign be with you!

Share / Save