Perl Beginners' Site

Perl - because programming should be fun.

Home → Online Tutorials → The "Perl for Newbies" Tutorial → Part 1

- 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 1 - The Perl Beginners' Site

Learn Perl Now!
And get a job doing Perl.

Show Navigation Controls

"Perl for Perl Newbies" - Part 1 ¶

Contents

Licence ¶

1. Introduction ¶

"There's more than one way to do it." **(The Perl Motto)**

1.1. The capabilities of Perl ¶

- Strings and data structures that are unlimited in size, nested to any depth.
- Powerful syntax and built-in functions.
- Extended, built-in support for regular expressions.
- Support for namespaces, classes, and objects.
- Functional Programming capabilities such as closures and continuations.
- CPAN - a comprehensive on-line archive of easily installable and re-usable modules.
- Powerful debugger with optional visual front-ends.
- Runs on almost any platform imaginable.
- Relatively portable and secure code.

1.2. A brief history of Perl ¶

Perl was introduced in 1987 (4 years before Linux itself), when the author, **Larry Wall**, released version 1.000 of it. The reason for its creation was that Wall was unhappy by the functionality that sed, C, awk and the Bourne Shell offered him. He looked for a language that will combine all of their best features, while having as few disadvantages of its own.

Since then, perl has seen several versions, each adding additional functionality. perl version 5, which was released in 1994, was a complete re-write of the perl interpreter, and introduced such things as hard references, modules, objects and lexical scoping. Several second-digit versions of perl 5 appeared since then, and the most up-to-date stable version (as of November 2010) is 5.12.x.

Perl became especially popular as a language for writing server-side scripts for web-servers. But that's not the only use of perl, as it is commonly used for system administration tasks, managing database data, as well as writing GUI applications.

Links:

- The Perl Timeline.
- Perl's history on the Perl 5 Wiki.
- *The Taming of the Camel* - a lecture by Larry Wall about the Evolution of Perl.

1.3. The perl development cycle ¶

Perl is interpreted, so no compilation is needed. To use perl, one should create a text file that contains the Perl program. This file can be written using any text editor available on your system. It is recommended that you end the filenames of your perl scripts with ".pl" in order to designate them as perl scripts.

After you are done, you should invoke the perl interpreter with the name of the file you created. Assuming your file is name "myscript.pl", you should type:

```
$ perl myscript.pl
```

At the command line (and press Enter). If your program contains errors or warnings they will be displayed. If your program does not contain errors, it will be executed.

The fact that the program was executed does not mean it does what you want it to do. We will learn how to debug Perl programs, later on in this series.

2. Basic Output (The "Hello World" program) ¶

In Perl we use the `print` command to echo strings and expressions to the screen.

The neophyte "Hello World!" program can be written in perl as follows:

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

Now here's some explanation (in case you need any):

1. The string is enclosed in double-quotes(`" ... "`) because that's how string constants are represented in perl.
2. `\n` is a special character that is called "newline". When you print a newline you will see that your output started on a new line.
3. The semi-colon (`;`) at the end indicates that this is the end of a perl command. Every perl command should be terminated with a semicolon.

2.1. More about semicolons ¶

A perl program can naturally include more than one command. To do so, you need to place a semicolon at the end of each command or otherwise the interpreter will be confused.

It is not enough to put each command on a separate line. In fact it's not even necessary. The following two programs are equivalent:

```
print "One Fish,\n";
print "Two Fish,\n";
print "Red Fish,\n";
print "Blue Fish.\n";
```

and

```
print "One Fish,\n"; print "Two Fish,\n";
print "Red Fish,\n"; print "Blue Fish.\n";
```

However, for readability and easy debugging it is recommended that each statement will be on a separate line. And sometimes it's helpful to span one on more than one line.

3. Expressions ¶

Perl supports such mathematical operators as `+`, `-`, `*` (multiplication), and parenthesis (`( ... )`). An example is worth a thousand words:

```
print "5 + 6 = ", 5+6, "\n";
print "(2 + 3) * 6 = ", (2+3)*6, "\n";
print "2 + 3 * 6 = ", 2+3*6, "\n";
print "2 raised to the power of 8 is ", 2**8, "\n";
print "10-5 = ", 10-5, ". 5-10 = ", 5-10, "\n";
print "2/3 = ", 2/3, "\n";
```

The output of this program is:

```
5 + 6 = 11
(2 + 3) * 6 = 30
2 + 3 * 6 = 20
2 raised to the power of 8 is 256
10-5 = 5. 5-10 = -5
2/3 = 0.666666666666667
```

The operators have the same precedence as their mathematical equivalents. The parenthesis are useful for making sure a sub-expression will be evaluated before all others do.

Note:

You can use commas to print more than one expression at once. It beats writing a separate `print` command for every expression you wish to output. However, for better readability, it is recommended that you will separate your expressions among several prints.

3.1. Operators and Precedence ¶

Here are some perl operators of interest.

+ , - , * , /

Respectively adds, subtracts, multiplies and divides two floating point numbers.

a ** b

Raises "a" to the power of "b". Works on floating point numbers too.

a . b

Concatenates two strings. The comma (`,`) as used by print does not really concatenates two strings, but rather prints them one after the other. (There's a subtle difference in functionality of the print command too, but we won't get into that, now).

```
print "Hello," . " " . "World!" . "\n" .
    "And this is the second line.\n";
```

a % b

Returns the modulo (remainder) of "b" from "a". If "a" and "b" are not integers they are rounded to an integral value.

(sub-expr)

Makes sure that `sub-expr` is evaluated as a separate sub-expression , an operation that could override the default operator precedence.

There are many more, but they will be covered later. For a complete list and more detailed information about the various perl operators consult the ["perlop" document] on your system.

3.2. Functions ¶

How many characters are in the perl motto? Perl can tell that right away:

```
print length("There's more than one way to do it"), "\n";
```

length() is a built-in function that tells how many characters are in a string. A function is a named sub-routine that accepts several arguments and returns a value that can be further evaluated as part of a greater expression, or used directly.

To help us understand functions further, let's inspect the perl function substr (short for "substring"). substr retrieves sub-strings of a given string. The first argument to it is a string, the second is the offset from which to take the substring, and the third is the length of the substring to be taken. The third one is optional and if unspecified, returns the substring till the end of the string.

An example will illustrate it best:

```
print substr("A long string", 3), "\n";
print substr("A long string", 1, 4), "\n";
print substr("A long string", 0, 6), "\n";
```

The output of this program is:

```
ong string
 lon
A long
```

(You may notice that the position of the first character is 0.)

The commas are used to separate the arguments to the function and they are mandatory in perl. The parenthesis that enclose them are optional, though. The above program could have been re-written as:

```
print ((substr "A long string", 3), "\n");
print ((substr "A long string", 1, 4), "\n");
print ((substr "A long string", 0, 6), "\n");
```

We need an extra set of parenthesis so print (which is also a function) would not be confused and consider only the result of the substr operation as its argument. If it makes no sense, then it shouldn't; however, remember that a set of parenthesis, that wraps up the argument list of a function, can do you no harm.

The int() function

Another useful function is int(). This function takes a number and rounds it down to a near integer (= whole number). Here's an example:

```
print "The whole part of 5.67 is " . int(5.67) . "\n";
```

3.3. More about strings ¶

In perl, strings and numbers are seamlessly converted into each other depending on the context in which they are used.

```
print (("5" + "6"), "\n");
print ((56 . 23), "\n");
```

The output of this program is:

```
11
5623
```

3.3.1. Escape Sequences ¶

We have already encountered the \n "escape sequence" which can come inside strings and designates a newline character. There are many others in perl. Here is a list of the most important ones:

- \\ - designates an actual backslash (\)
- \" - designates an actual double-quote character (")
- \$ - an actual dollar sign (a real $ does something else)
- \@ - an actual at-sign (a non-escaped @ does something else)
- \n - a newline character
- \r - a carriage return sign
- \t - a tab character
- \xDD - where "DD" are two hexadecimal digits - gives the character whose ASCII code is "DD".

Here's an example to illustrate some of them:

```
print "I said \"hi!\" to myself, and received no reply.\n";

print "This program will cost you \$100 dollars.\n";

print "The KDE\\GNOME holy war makes life in the Linux world " .
      "more interesting.\n";
```

whose output is:

```
I said "hi!" to myself, and received no reply.
This program will cost you $100 dollars.
The KDE\GNOME holy war makes life in the Linux world more interesting.
```

4. Variables ¶

Variables are named cells stored in the computer memory that can hold any single Perl value. One can change the value that a variable holds, and one can later retrieve the last value assigned as many times as wanted.

Variables in Perl start with a dollar sign ($) and proceed with any number of letters, digits and underscores (_) as long as the first letter after the dollar is a letter or underscore.

To retrieve the value of a variable one simply places the variable name (again including the dollar sign) inside an expression.

To assign value to a variable, one places the full variable name (including the dollar sign) in front of an equal sign (=) and places the value to the right of the equal sign. This form is considered a statement and should be followed by a semicolon. The value assigned may be an expression that may contain other variables (including the assigned variable itself!).

An example will illustrate it:

```perl
$myvar = 17;
$x = 2;
print $myvar, " * ", $x, " = " , ($myvar*$x), "\n";
$x = 10;
print $myvar, " * ", $x, " = " , ($myvar*$x), "\n";
$x = 75;
print $myvar, " * ", $x, " = " , ($myvar*$x), "\n";
$x = 24;
print $myvar, " * ", $x, " = " , ($myvar*$x), "\n";
```

The output of this program is:

```
17 * 2 = 34
17 * 10 = 170
17 * 75 = 1275
17 * 24 = 408
```

Several things can be noticed:

1. The value of $x changes throughout the program. It's perfectly fine, and usually even necessary to modify the value of a variable.

2. By using `$myvar` we can ensure, that assuming we wish to change its value, we will only have to change it in one place, not in every place it appears.

4.1. "+=" and friends ¶

Perl provides a shortcut for writing "$myvar = $myvar + $value", or "$myvar = $myvar / $value" and similar operations. Here's an example:

```
$x = 1;
$y = 0;
print "2^", $y, "=", $x, "\n";
$x *= 2; $y += 1;
print "2^", $y, "=", $x, "\n";
$x *= 2; $y += 1;
print "2^", $y, "=", $x, "\n";
$x *= 2; $y += 1;
print "2^", $y, "=", $x, "\n";
$x *= 2; $y += 1;
print "2^", $y, "=", $x, "\n";
$x *= 2; $y += 1;
print "2^", $y, "=", $x, "\n";
```

Since the operations `$x += 1` and `$x -= 1` are so commonly used, they were also assigned a separate operator. One can use `$x++` and `$x--` to perform them. For example, the above program could have been written as:

```
$x = 1;
$y = 0;
print "2^", $y, "=", $x, "\n";
$x *= 2; $y++;
print "2^", $y, "=", $x, "\n";
$x *= 2; $y++;
print "2^", $y, "=", $x, "\n";
$x *= 2; $y++;
print "2^", $y, "=", $x, "\n";
$x *= 2; $y++;
print "2^", $y, "=", $x, "\n";
$x *= 2; $y++;
print "2^", $y, "=", $x, "\n";
```

5. Input ¶

In order to receive a value from the user, perl supplies the `<>` operator. When entered, this operator reads a line from the command line, and returns it (along with the newline character).

Here's an example:

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

Notice the `chomp` function which strips off the trailing newline character from the variable. You would usually want to use it, when getting input from the user.

Here's another example:

```
print "Please enter a string:";
$string = <>;
chomp($string);
print "The string you entered contains ", length($string), " characters.\n";
```

6. The For Loop ¶

The for loop enables us to iterate over a sequence of numbers and repeat the same set of operations for each number.

For example, the following program prints all the numbers from 1 to 100:

```
for $i (1..100)
{
    print $i, "\n";
}
```

Some explanations about the syntax:

1. `$i` is the iteration variable. It receives the value 1, then the value 2, then 3 and so forth until it is equal to 100, afterwards the loop terminates.
2. The curly brackets (`{ ... }`) encapsulate the loop block. The loop block is executed once for each value `$i` accepts. Within that block, called the loop body, you can use `$i` and you'll get its current value.

We can nest loops, so for example the following program prints the multiplication board:

```
for $y (1 .. 10)
{
    for $x (1 .. 10)
    {
        $product = $y*$x;
        # Add as much whitespace as needed so the number will occupy
        # exactly 4 characters.
        for $whitespace (1 .. (4-length($product)))
        {
            print " ";
        }
        print $product;
    }
    # Move to the next line
    print "\n";
}
```

You may have noticed the program's comments. In perl comments start with the sharp sign (#) and extend to the end of the line. Writing the multiplication boards with the labels that indicate which numbers are being multiplied is left as an exercise to the reader.

7. Conditionals ¶

Conditionals enable us to execute a group of statements if a certain condition is met. For example the following program, reports to the user whether or not his name starts with the letter "A":

```
print "Please enter your name:\n";
$name = <>;
if (substr($name,0,1) eq "A")
{
    print "Your name starts with 'A'!\n";
}
else
{
    print "Your name does not start with 'A'!\n";
}
```

The code `substr($name,0,1) eq "A"` is a condition expression which uses the perl `eq` operator, which returns true if and only if the strings are the same. There are more such operators and they will be explained shortly.

Inside the curly brackets following the `if` there is the code to be executed by the conditional. That code can be as long as you like. The else part is executed assuming the conditional was found to be false, and is optional.

7.1. Numerical Comparison Operators ¶

Perl supplies the user with 6 numerical comparison operators which test for the comparison of two numbers. They are:

`$x == $y`	`$x` and `$y` are equal.
`$x > $y`	`$x` is greater than `$y`

`$x < $y`	`$x` is lesser than `$y`
`$x >= $y`	`$x` is greater or equal to `$y`
`$x <= $y`	`$x` is lesser or equal to `$y`
`$x != $y`	`$x` is not equal to `$y`

Those operators can be used inside conditionals and also outside, as part of a normal expression. The following program prints all the numbers between 1 and 100 that are not divisible by 3:

```
for $n (1 .. 100)
{
    if (($n % 3) != 0)
    {
        print $n, "\n";
    }
}
```

7.2. String Comparison Operators ¶

In order to compare for string equality, or if one string is alphabetically bigger than another, you can use the six string comparison operators. Here are the string operators together with the numerical operators they correspond too:

String Operator	Numerical Operator
eq	==
ne	!=
gt	>
lt	<
ge	>=
le	<=

Notice that the string operators are built from the initials of their abbreviated names. (E.g: eq = equal, gt = greater than). Perl's string comparison is case-sensitive. If you want a case insensitive string comparison, use the `lc` function to convert the strings to lowercase beforehand.

Example:

```
print "Please enter your private name:\n";
$name = <>;
chomp($name);
if (lc($name) eq "rachel")
{
    print "Your name is Rachel!\n";
}
else
{
    print "Your name is not Rachel!\n";
}
```

7.3. Boolean Operators ¶

Sometimes it is useful to check for the validation of more than one condition. For doing this, Perl supplies boolean operators.

&&

`$x && $y` evaluates to true if both `$x` and `$y` are true. It is called the “logical and” of the two operands `$x` and `$y`.

||

`$x || $y` (called the “logical or” of `$x` and `$y`) is evaluated to true if one or both of its operands are true.

!

`! $x` (pronounced “not `$x`”) evaluates to true if `$x` is false.

Note that if the first operand is evaluated to false in an `&&` operation, the second operand will not be evaluated at all. Similarly, if the first operand of `||` was found to be true, the second one will not be evaluated either.

If you wish both operands to be evaluated at all times you need to assign them to variables first.

Here are some examples:

```
print "Please enter the lower bound of the range:\n";
$lower = <>;
chomp($lower);
print "Please enter the upper bound of the range:\n";
$upper = <>;
chomp($upper);
if ($lower > $upper)
{
    print "This is not a valid range!\n";
}
else
{
    print "Please enter a number:\n";
    $number = <>;
    chomp($number);
    if (($lower <= $number) && ($number <= $upper))
    {
        print "The number is in the range!\n";
    }
    else
    {
        print "The number is not in the range!\n";
    }
}
```

```
print "Please enter your name:\n";
$name = <>;
chomp($name);
$fl = lc(substr($name, 0, 1));
if (($fl eq "a")||($fl eq "b")||($fl eq "c"))
{
    print "Your name starts with one of the " .
        "first three letters of the ABC.\n";
}
else
{
    print "Your name does not start with one of the " .
        "first three letters of the ABC.\n";
}
```

Note: The function `lc()` converts a string to lowercase.

7.4. True Expressions vs. False Expressions ¶

In Perl every expression is considered true except for the following three cases:

1. The number `0`.
2. The empty string (`""`).
3. A special value called `undef`. This is the default value of every variable that was not initialized before it was accessed.

One of the most convenient ways of cancelling code is to wrap it in an `if (0) { ... }` block. It usually is faster than adding comments to the beginning of each line. Of course, you should not distribute code that way, but it is useful for testing conjectures.

7.5. elsif ¶

The full syntax of the if statement is as follows:

```
if ($condition0)
{
}
elsif ($condition1)
{
}
elsif ($condition2)
{
}
.
.
.
else
{
}
```

The N-th elsif block will be executed only if all the conditions in the previous blocks were not met **and** its own condition was met.

You can do the same with nesting if statements inside else statements, but isn't it nice of perl to save you all those extra brackets and indentations?

8. The While Loop ¶

The while loop enables you to repeat a sequence of statements an arbitrary number of times for as long a condition is met. The syntax is very similar to the if statement and will be introduced in the following example:

```
print "Please enter a number:\n";
$number=<>;
chomp($number);
$power_of_2 = 1;
while ($power_of_2 < $number)
{
    $power_of_2 *= 2;
}
print ("The first power of 2 that is " .
    "greater than this number is " , $power_of_2, "\n");
```

It is possible that a while loop will not be executed at all, if the condition is not met right on the start.

The following program checks if a given string is made entirely of "a" characters:

```
print "Please enter a string:\n";
$string=<>;
chomp($string);

# Initialize all_as to TRUE
$all_as = 1;

# The first position in the string.
$position = 0;

while ($all_as && ($position < length($string)))
{
    $char = lc(substr($string, $position, 1));

    if ($char ne "a")
    {
        $all_as = 0;
    }

    # Increment the position
    $position++;
}

if ($all_as)
{
    print "The string you entered is all A's!\n";
}
else
{
    print "At least one of the characters in the string " .
        "you entered is not \"A\".\n";
}
```

8.1. last and next ¶

Within the `while` and `for` loops one can use two special commands, called `last` and `next`. `last` terminates a loop prematurely while `next` skips the rest of the remaining loop body, skips to the loop condition and if it is met, executes the loop again.

By default, `last` and `next` operate on the most innermost loop. However, the loop to which they relate can be controlled by labelling the requested loop and specifying this label as a parameter to `last` or `next`.

The following example is a rewrite of the "All A's" program using `last`:

```
print "Please enter a string:\n";
$string=<>;
```

```
chomp($string);

# The first position in the string.
$position = 0;

while ($position < length($string))
{
    $char = lc(substr($string, $position, 1));

    if ($char ne "a")
    {
        last;
    }

    # Increment the position
    $position++;
}

# If the position is the end of the string it means the loop was not
# terminated prematurely, so an "a" was not encountered.
if ($position == length($string))
{
    print "The string you entered is all A's!\n";
}
else
{
    print "At least one of the characters in the string " .
        "you entered is not \"A\".\n";
}
```

This program prints a left-tilted pyramid:

```
print "Please enter the length of the pyramid:\n";
$size = <>;
chomp($size);

ROW_LOOP: for $row (1 .. $size)
{
    for $column (1 .. ($size+1))
    {
        if ($column > $row)
        {
            print "\n";
            next ROW_LOOP;
        }
        print "#";
    }
}
```

"ROW_LOOP" is the label for the outer loop, and it can be seen that next uses it as a parameter. All in all, next and last are sometimes very convenient (but don't tell it to **Edsger W. Dijkstra**'s face!), so you will see them being used often.

9. Arrays ¶

Arrays are a sequence of variables, whose members can be retrieved and assigned to by using their indices. An index passed to an array may well be, and usually is, another variable.

To refer to the $i'th element of the array @myarray, one uses the syntax $myarray[$i]. This element can be assigned to or its value can be retrieved, with the same notation.

Array indices are whole numbers and the first index is 0. As in the length of a string, the number of elements in an array is bounded only by the amount of available memory the computer has.

The following program prints the primes up to 200:

```perl
$num_primes = 0;

# Put 2 as the first prime so we won't have an empty array,
# what might confuse the interpreter
$primes[$num_primes] = 2;
$num_primes++;

MAIN_LOOP:
for $number_to_check (3 .. 200)
{
    for $p (0 .. ($num_primes-1))
    {
        if ($number_to_check % $primes[$p] == 0)
        {
            next MAIN_LOOP;
        }
    }

    # If we reached this point it means $number_to_check is not
    # divisible by any prime number that came before it.
    $primes[$num_primes] = $number_to_check;
    $num_primes++;
}

for $p (0 .. ($num_primes-1))
{
    print $primes[$p], ", ";
}
print "\n";
```

The notation scalar(@myarray) can be used to refer to the number of elements in an array. This number is equal to the maximal index which was assigned in the array plus one. You will also see the notation $#myarray which is equal to the maximal index itself (or -1 if the array is empty).

Thus, for example, the above program could have been written as follows:

```
# Put 2 as the first prime so we won't have an empty array,
# what might confuse the interpreter
$primes[0] = 2;

MAIN_LOOP:
for $number_to_check (3 .. 200)
{
    for $p (0 .. $#primes)
    {
        if ($number_to_check % $primes[$p] == 0)
        {
            next MAIN_LOOP;
        }
    }

    # If we reached this point it means $number_to_check is not
    # divisible by any prime number that came before it.
    $primes[scalar(@primes)] = $number_to_check;
}

for $p (0 .. $#primes)
{
    print $primes[$p], ", ";
}
print "\n";
```

9.1. The ',' operator ¶

In perl the comma (,) is an operator, which we encountered before in function calls. The comma concatenates two arrays. We can use it to initialise an array in one call:

```
@lines = ("One fish", "Two fish", "Red fish", "Blue fish");

for $idx (0 .. $#lines)
{
    print $lines[$idx], "\n";
}
```

We can also use it to concatenate two existing arrays:

```
@primes1 = (2,3,5);
@primes2 = (7,11,13);
@primes = (@primes1,@primes2,17);
@primes = (@primes,19);

for $idx (0 .. $#primes)
{
    print $primes[$idx], "\n";
}
```

So why it is used in function calls? In perl every function accepts an array of arguments and returns an array of return values. That's why the comma is useful for calling functions which accept more than one argument.

9.2. Negative Indexes ¶

The expression `$myarray[-$n]` is equivalent to `$myarray[scalar(@myarray)-$n]`. I.e: subscripts with negative indexes return the `$n`'th element from the end of the array. So to get the value of the last element you can write `$myarray[-1]` and for the second last `$myarray[-2]`, etc.

Note that one should also make sure that array subscripts that are continuously decremented will not underflow below 0, or else one will start getting the elements from the end of the array.

9.3. The foreach loop ¶

By using the `foreach` loop we can iterate over all the elements of an array, and perform the same set of operations on each one of them. Here's an example:

```
@numbers = (15,5,7,3,9,1,20,13,9,8,
    15,16,2,6,12,90);

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

foreach $i (@numbers[1..$#numbers])
{
    if ($i > $max)
    {
        $max = $i;
    }
    elsif ($i < $min)
    {
        $min = $i;
    }
}
```

```
print "The maximum is " . $max . "\n";
print "The minimum is " . $min . "\n";
```

The foreach loop in the example assigns each of the elements of the array which was passed to it to $i in turn, and executes the same set of commands for each value.

9.3.1. The for keyword and the .. operator ¶

The for keyword in Perl means exactly the same as foreach and you can use either one interchangeably.

$x .. $y is a special operator that returns an array containing the sequence of consecutive integers from $x up to and including $y. Now one can fully understand the for $i (1 .. 10) construct that we used in the beginning of this lecture.

9.4. Built-In Array Functions ¶

push

The push function appends an element or an entire array to the end of an array variable. The syntax is push @array_to_append_to, @array_to_append or push @array, $elem1. For example, the primes program from earlier could be written as:

```
# Put 2 as the first prime so we won't have an empty array,
# what might confuse the interpreter
@primes = (2);

MAIN_LOOP:
for $number_to_check (3 .. 200)
{
    foreach $p (@primes)
    {
        if ($number_to_check % $p == 0)
        {
            next MAIN_LOOP;
        }
    }

    # If we reached this point it means $number_to_check is not
    # divisible by any prime number that came before it.
    push @primes, $number_to_check;
}

foreach $p (@primes)
{
    print $p, ", ";
}
print "\n";
```

Notice that push is equivalent to typing @array = (@array, $extra_elem), but it is recommended to use it, because it minimises error and it executes faster.

pop

pop extracts the last element from an array and returns it. Here's a short example to demonstrate it:

```
# This program prints the numbers from 10 down to 1.
@numbers = (1 .. 10);
while(scalar(@numbers) > 0)
{
    $i = pop(@numbers);
    print $i, "\n";
}
```

shift

shift extracts the **first** element of an array and returns it. The array will be changed to contain only the elements that were present there previously, with the 1 to scalar(@array)-1 indexes.

Here's the above example, while using shift instead of pop:

```
# This program prints the numbers 1 to 10.
@numbers = (1 .. 10);
while(scalar(@numbers) > 0)
{
    $i = shift(@numbers);
    print $i, "\n";
}
```

join

The syntax is join($separator, @array) and what it does is concatenates the elements of @array while putting $separator in between. Here's an example:

```
@myarray = ("One fish", "Two fish", "Red Fish", "Blue Fish");

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

reverse

The reverse function returns the array which contains the elements of the array passed to it as argument in reverse. Here's an example:

```
print "Enter some lines:\n";

$line = <>;
chomp($line);
while ($line)
{
    push @mylines, $line;
    $line = <>;
    chomp($line);
}

print "Your lines in reverse are:\n", join("\n", reverse(@mylines)), "\n";
```

Note that by typing scalar(reverse($scalar)) you get the string that contains the characters of $scalar in reverse. scalar(reverse(@array)) concatenates the array into one string and then reverses its characters.

9.5. The x operator ¶

The expression (@array) x $num_times returns an array that is composed of $num_times copies of @array one after the other. The expression $scalar x $num_times, on the other hand, returns a string containing $num_times copies of $scalar concatenated together string-wise.

Therefore it is important whether the left operand is wrapped in parenthesis or not. It is usually a good idea to assign the left part to a variable before using x so you'll have the final expression ready.

Here's an example to illustrate the use:

```
print "Test 1:\n";
@myarray = ("Hello", "World");
@array2 = ((@myarray) x 5);
print join(", ", @array2), "\n\n";

print "Test 2:\n";
@array3 = (@myarray x 5);
print join(", ", @array3), "\n\n";

print "Test 3:\n";
$string = "oncatc";
print (($string x 6), "\n\n");

print "Test 4:\n";
print join("\n", (("hello") x 5)), "\n\n";
```

```perl
print "Test 5:\n";
print join("\n", ("hello" x 5)), "\n\n";
```

Can you guess what the output of this program will be?

Here's a spoiler

```

Test 1:
Hello, World, Hello, World, Hello, World, Hello, World, Hello, World

Test 2:
22222

Test 3:
oncatconcatconcatconcatconcatconcatc

Test 4:
hello
hello
hello
hello
hello

Test 5:
hellohellohellohellohello
```

Share / Save