Perl Programming

Wikibooks.org
On the 28th of April 2012 the contents of the English as well as German Wikibooks and Wikipedia projects were licensed under Creative Commons Attribution-ShareAlike 3.0 Unported license. An URI to this license is given in the list of figures on page 131. If this document is a derived work from the contents of one of these projects and the content was still licensed by the project under this license at the time of derivation this document has to be licensed under the same, a similar or a compatible license, as stated in section 4b of the license. The list of contributors is included in chapter Contributors on page 125. The licenses GPL, LGPL and GFDL are included in chapter Licenses on page 135, since this book and/or parts of it may or may not be licensed under one or more of these licenses, and thus require inclusion of these licenses. The licenses of the figures are given in the list of figures on page 131. This PDF was generated by the \LaTeX typesetting software. The \LaTeX source code is included as an attachment (source.7z.txt) in this PDF file. To extract the source from the PDF file, we recommend the use of http://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/ utility or clicking the paper clip attachment symbol on the lower left of your PDF Viewer, selecting Save Attachment. After extracting it from the PDF file you have to rename it to source.7z. To uncompress the resulting archive we recommend the use of http://www.7-zip.org/. The \LaTeX source itself was generated by a program written by Dirk H"{a}ijm"{a}ger, which is freely available under an open source license from http://de.wikibooks.org/wiki/Benutzer:Dirk_Huenniger/wb2pdf. This distribution also contains a configured version of the pdflatex compiler with all necessary packages and fonts needed to compile the \LaTeX source included in this PDF file.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1 Section 1: Beginning Perl</td>
<td>1</td>
</tr>
<tr>
<td>0.2 Getting Started</td>
<td>1</td>
</tr>
<tr>
<td>0.3 A First Taste of Perl</td>
<td>3</td>
</tr>
<tr>
<td>0.4 Strings</td>
<td>4</td>
</tr>
<tr>
<td>0.5 Single Quoted Strings</td>
<td>5</td>
</tr>
<tr>
<td>0.6 Brief Digression from Strings Alone: The <code>print</code> Function</td>
<td>7</td>
</tr>
<tr>
<td>0.7 Double Quoted Strings</td>
<td>9</td>
</tr>
<tr>
<td>0.8 Numbers</td>
<td>11</td>
</tr>
<tr>
<td>0.9 Simple variables</td>
<td>13</td>
</tr>
<tr>
<td>0.10 Arrays</td>
<td>14</td>
</tr>
<tr>
<td>0.11 Associative arrays</td>
<td>14</td>
</tr>
<tr>
<td>0.12 Subroutines</td>
<td>15</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>17</td>
</tr>
<tr>
<td>2 The operators</td>
<td>19</td>
</tr>
<tr>
<td>2.1 Arithmetic</td>
<td>19</td>
</tr>
<tr>
<td>2.2 Assignment</td>
<td>20</td>
</tr>
<tr>
<td>2.3 Comparison</td>
<td>21</td>
</tr>
<tr>
<td>2.4 Logical</td>
<td>21</td>
</tr>
<tr>
<td>2.5 Bitwise</td>
<td>23</td>
</tr>
<tr>
<td>2.6 String</td>
<td>23</td>
</tr>
<tr>
<td>2.7 Comparing strings</td>
<td>23</td>
</tr>
<tr>
<td>2.8 File Test</td>
<td>23</td>
</tr>
<tr>
<td>2.9 Other</td>
<td>24</td>
</tr>
<tr>
<td>2.10 Precedence</td>
<td>24</td>
</tr>
<tr>
<td>2.11 The smart match operator (```)</td>
<td>25</td>
</tr>
<tr>
<td>2.12 Dereferencing</td>
<td>25</td>
</tr>
<tr>
<td>2.13 Scalar Variables</td>
<td>26</td>
</tr>
<tr>
<td>2.14 Lists</td>
<td>29</td>
</tr>
<tr>
<td>2.15 Arrays</td>
<td>31</td>
</tr>
<tr>
<td>2.16 Command line arguments</td>
<td>33</td>
</tr>
<tr>
<td>2.17 Related Articles</td>
<td>33</td>
</tr>
<tr>
<td>2.18 &quot;Associative Arrays&quot;</td>
<td>35</td>
</tr>
<tr>
<td>2.19 Working with hashes</td>
<td>35</td>
</tr>
<tr>
<td>2.20 Hash of Hashes of Hashes</td>
<td>35</td>
</tr>
<tr>
<td>2.21 Output</td>
<td>36</td>
</tr>
<tr>
<td>2.22 Input</td>
<td>36</td>
</tr>
<tr>
<td>3 Unit Exercise</td>
<td>39</td>
</tr>
<tr>
<td>3.1 Advanced Output Overview</td>
<td>39</td>
</tr>
<tr>
<td>3.2 Use other quotes</td>
<td>39</td>
</tr>
<tr>
<td>3.3 Escape characters</td>
<td>39</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>3.4</td>
<td>Custom Quotes</td>
</tr>
<tr>
<td>3.5</td>
<td>Block Output</td>
</tr>
<tr>
<td>3.6</td>
<td>Variable Output</td>
</tr>
<tr>
<td>3.7</td>
<td>Caveats</td>
</tr>
<tr>
<td>3.8</td>
<td>Control structures</td>
</tr>
<tr>
<td>3.9</td>
<td>See Also</td>
</tr>
<tr>
<td>3.10</td>
<td>Control structures</td>
</tr>
<tr>
<td>3.11</td>
<td>See Also</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Read files</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>Procedural Interface</td>
<td>45</td>
</tr>
<tr>
<td>4.2</td>
<td>Object Oriented Interface</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>String functions</td>
<td>46</td>
</tr>
<tr>
<td>4.4</td>
<td>Numeric functions</td>
<td>52</td>
</tr>
<tr>
<td>4.5</td>
<td>Array functions</td>
<td>54</td>
</tr>
<tr>
<td>4.6</td>
<td>List functions</td>
<td>55</td>
</tr>
<tr>
<td>4.7</td>
<td>Associative array functions</td>
<td>56</td>
</tr>
<tr>
<td>4.8</td>
<td>Input and output functions</td>
<td>57</td>
</tr>
<tr>
<td>4.9</td>
<td>Functions for working with fixed length records</td>
<td>60</td>
</tr>
<tr>
<td>4.10</td>
<td>Filesystem functions</td>
<td>61</td>
</tr>
<tr>
<td>4.11</td>
<td>Program functions</td>
<td>65</td>
</tr>
<tr>
<td>4.12</td>
<td>Misc functions</td>
<td>67</td>
</tr>
<tr>
<td>4.13</td>
<td>Processes</td>
<td>69</td>
</tr>
<tr>
<td>4.14</td>
<td>Modules</td>
<td>70</td>
</tr>
<tr>
<td>4.15</td>
<td>Classes and objects</td>
<td>70</td>
</tr>
<tr>
<td>4.16</td>
<td>Sockets</td>
<td>72</td>
</tr>
<tr>
<td>4.17</td>
<td>Login information</td>
<td>72</td>
</tr>
<tr>
<td>4.18</td>
<td>Network information</td>
<td>72</td>
</tr>
<tr>
<td>4.19</td>
<td>Time and date</td>
<td>72</td>
</tr>
<tr>
<td>4.20</td>
<td>Functions that reverse each other</td>
<td>74</td>
</tr>
<tr>
<td>4.21</td>
<td>Section 2: In-depth Perl ideas</td>
<td>75</td>
</tr>
<tr>
<td>4.22</td>
<td>Introduction</td>
<td>75</td>
</tr>
<tr>
<td>4.23</td>
<td>Referencing and Dereferencing Syntax</td>
<td>75</td>
</tr>
<tr>
<td>4.24</td>
<td>External links</td>
<td>75</td>
</tr>
<tr>
<td>4.25</td>
<td>Overview</td>
<td>80</td>
</tr>
<tr>
<td>4.26</td>
<td>Section 3: Interfacing Perl</td>
<td>84</td>
</tr>
<tr>
<td>4.27</td>
<td>Links</td>
<td>84</td>
</tr>
<tr>
<td>4.28</td>
<td>Installing modules</td>
<td>85</td>
</tr>
<tr>
<td>4.29</td>
<td>Using a module in your program</td>
<td>86</td>
</tr>
<tr>
<td>4.30</td>
<td>Finding documentation</td>
<td>86</td>
</tr>
<tr>
<td>4.31</td>
<td>Contributing your own modules to CPAN</td>
<td>87</td>
</tr>
<tr>
<td>4.32</td>
<td>Section 4: CGI and Apache</td>
<td>88</td>
</tr>
<tr>
<td>4.33</td>
<td>The Initial Setup</td>
<td>88</td>
</tr>
<tr>
<td>4.34</td>
<td>Retrieving Information</td>
<td>89</td>
</tr>
<tr>
<td>4.35</td>
<td>Output</td>
<td>89</td>
</tr>
<tr>
<td>4.36</td>
<td>The Finished Code</td>
<td>89</td>
</tr>
<tr>
<td>4.37</td>
<td>Frameworks</td>
<td>90</td>
</tr>
<tr>
<td>4.38</td>
<td>External links</td>
<td>91</td>
</tr>
<tr>
<td>4.39</td>
<td>The Initial Setup</td>
<td>91</td>
</tr>
<tr>
<td>4.40</td>
<td>Retrieving Information</td>
<td>91</td>
</tr>
<tr>
<td>4.41</td>
<td>Output</td>
<td>91</td>
</tr>
</tbody>
</table>
0.1 Section 1: Beginning Perl

0.2 Getting Started

This book assumes that you know absolutely nothing about programming at all and that Perl is your first language. However, basic operations such as making text files are outside of the realm of this tutorial.

0.2.1 Obtaining Perl

To find out if you already have Perl installed on your computer, go into the command line and type:

perl -v
This will display which version of Perl you have installed on your computer, if it is installed.

There are at least two easy ways to install Perl on Windows: the ActiveState\(^1\) distribution, and the Strawberry Perl\(^2\) distribution. Both are downloadable as native Windows installers. ActivePerl has a prebuilt package repository and is supported by a corporation, while Strawberry Perl includes a compiler (gcc) so that perl modules can be installed "on the fly" and is community-supported.

Most Unix-like operating systems will include Perl by default, and Linux Standard Base mandates that all compliant Linuxes ship with Perl installed. However, if for some reason you don't have perl, you can explore the options available to you at the main Perl download page\(^3\), which will provide links to source and binaries.

### 0.2.2 Writing programs

#### A Sample Program

Perl is an interpreted language, which means you will always need the Perl interpreter which will compile and execute your program each time you run it. Instead of compiling your program into bytecode like in Pascal\(^4\) or C++\(^5\) and then executing it, you can simply copy your program's source code to a different computer (that has the perl interpreter) and run it.

For our first example, run your favorite text editor, and type something like this:

```perl
#!/usr/bin/perl
use strict;
use warnings;
print "Hello World";
```

If you don't understand this yet, don't worry; This will be explained in more depth later.

Save the file as `myprog.pl` and you have a perl program ready to run.

### 0.2.3 Running programs

#### Windows

To run a perl program with a modern version of ActivePerl\(^6\) installed, you simply click on it. If the screen flashes and you can't see the output you might have to execute the file from within the Windows shell (ie. cmd.exe or PowerShell). With Strawberry Perl\(^7\), you'll have to execute a perl program from the command line as shown below.

---

1. [http://activestate.com/Products/ActivePerl/?_x=1](http://activestate.com/Products/ActivePerl/?_x=1)
2. [http://strawberryperl.com](http://strawberryperl.com)
3. [http://www.perl.com/download.csp](http://www.perl.com/download.csp)
From a windows command-line interface, you can run the program thusly:

```bash
C:\> perl path\to\foo\myprog.pl
```

or, if perl.exe is not in your path:

```bash
C:\> c:\perl\bin\perl.exe myprog.pl
```

Note: You may have to specify the full path to your program unless you are running the command prompt in that directory.

**UNIX-like Systems**

You can run a perl program by running perl itself, and telling the shell the name of the file:

```bash
perl myprog.pl
```

Usually, perl programs are made executable on their own. This involves two changes to the sample program. First, edit it and put the following shebang line\(^8\) at the top of the file:

```bash
#!/usr/bin/perl
```

Then, at a command prompt, make your program executable by using chmod.

```bash
chmod +x myprog.pl
```

Your program is now executable and ready to run, just like any other file. To execute, type:

```bash
./myprog.pl
```

By convention, .pl identifies a perl script, and .pm a perl library. The .pl file extension isn't needed for either of these examples; it's just a useful way of identifying files. The only time the convention should be violated is if the program is to be installed outside of the current working directory, and there runs a chance you might want to some day rewrite them in a different language.

### 0.3 A First Taste of Perl

Here's a simple program written in Perl to get us started:

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

# Outputs Hello world to the screen.
print "Hello world!\n";
```

\(^8\) [http://en.wikipedia.org/wiki/Shebang%20%28Unix%29](http://en.wikipedia.org/wiki/Shebang%20%28Unix%29)
Let's take a look at this program line by line:

- `#!/usr/bin/perl`

  On Unix systems this tells the Operating System to execute this file with the program located at `/usr/bin/perl`. This is the default Unix location for the perl interpreter, on Windows `#!C:\Perl\bin\perl.exe` or `#!C:\strawberry\perl\bin\perl.exe` (depending on whether ActivePerl or Strawberry Perl was installed) should be used instead.

  **Shebang:** A line at the start of a file, beginning with `#!`, that gives instructions to the operating system.

- `# Outputs ...`

  This line is a comment - it is ignored by the perl interpreter, but is very useful. It helps you to debug and maintain your code, and explain it to other programmers.

  **Comment:** A line of plain text ignored by the interpreter in a file of code.

- `print "Hello world!\n";`

  The `print` instruction writes whatever follows it to the screen. The `\n` at the end of the `string` puts a new line to the screen. The semicolon at the end of the line tells the perl interpreter that the instruction is finished; you must put a semicolon at the end of every instruction in Perl code.

  **String:** A sequence of characters used as data by a program.

### 0.3.1 Exercises

- Change the program so it says hello to `you`.
- Change the program so that after greeting you, it asks how you are doing, on the next line.
  The output should look like this:
  Hello your_name! How are you?

### 0.4 Strings

Any sequence of characters put together as one unit, is a string. So, the word `the` is a string. This sentence is a string. Even this entire paragraph is a string. In fact, you could consider the text of this entire book as one string.

Strings can be of any length and can contain any characters, numbers, punctuation, special characters (like `!`, `#`, and `%), and even characters in natural languages besides English. In addition, a string
can contain special whitespace formatting characters like newline, tab, and the bell character. We will discuss special characters more later on. For now, we will begin our consideration of strings by considering how to insert literal strings into a Perl program.

To begin our discussion of strings in Perl, we will consider how to work with string literals in Perl. The word literal here refers to the fact that these are used when you want to type a string directly to Perl. This can be contrasted with storing a string in a variable.

Any string literal can be used as an expression. We will find this useful when we want to store string literals in variables. However, for now, we will simply consider the different types of string literals that one can make in Perl. Later, we will learn how to assign these string literals to variables in the Scalar Variables section.

### 0.5 Single Quoted Strings

String literals can be represented in primarily three ways in Perl. We have already used one type in the simple programming examples, using double quote marks. Using double or single quote marks in Perl each has a special meaning.

Single quotes can be thought of as literal strings. In the previous examples, you may have noticed that variable names were included inside the strings with double quotes. When the results were printed, the value of the variable was placed in the printed line, not the name of the variable. If single quote marks were used, the actual variable name would have been printed because nearly all special characters that might be interpreted differently are taken at face value when using single quotes.

To see what is meant by this, try this simple program:

```perl
my $name = "Fred";
print "Hello $name\n";
print 'Hello $name\n';
```

You should see "Hello Fred" on the first line and "Hello $name\n" on the second (without a newline after it). Putting the value of $name into the string in the first print statement is called "interpolation." If you don't need interpolation, you should use single quotes, because it makes your intent clearer.

#### 0.5.1 Special Characters in Single-quoted Strings

There are two characters in single quoted strings that do not always represent themselves. This is due to necessity, since single-quoted strings start and end with the ' character. We need a way to express inside a single-quoted string that we want the string to contain a ' character.

The solution to this problem is to preceded any ' characters we actually want to appear in the string itself with the backslash (\ character). Thus we have strings like this:

```
,xxx\,xxx,;  # xxx, a single-quote character, and then xxx
```

We have in this example a string with 7 characters exactly. Namely, this is the string: xxx'xxx. It can be difficult at first to become accustomed to the idea that two characters in the input to Perl
actually produce only one character in the string itself. (C programmers are already probably used to this idea.) However, just keep in mind the rules and you will probably get used to them quickly.

Since we have used the `\` character to do something special with the `'` character, we must now worry about the special cases for the backslash character itself. When we see a `\` character in a single-quoted string, we must carefully consider what will happen.

Under most circumstances, when a `\` is in a single-quoted string, it is simply a backslash, representing itself, as most other characters do. However, the following exceptions apply:

- The sequence `\'` yields the character `'` in the actual string. (This is the exception we already discussed above).
- The sequence `\\` yields the character `\` in the actual string. In other words, two backslashes right next to each other actually yield only one backslash.
- A backslash, by itself, cannot be placed at the end of a the single-quoted string. This cannot happen because Perl will think that you are using the `\` to escape the closing `'`.

The following examples exemplify the various exceptions, and use them properly:

```
, I don\', t think so , ;       # Note the , inside is escaped with \\
, Need a \ (backslash) or \\?, ; # The \\ gives us \\
, You can do this: \\, ;       # A single backslash at the end
, Three \\, s: "\\\\" ;   # There are three \ chars between ""
```

In the last example, note that the resulting string is `Three ' s: "\\\\"`. If you can follow that example, you have definitely mastered how single-quoted strings work!

Instead of unreadable backslash escapes, Perl offers other ways of quoting strings. The first example above could be written as:

```
q{ I don\', t think so };     # No \ needed to escape the ,
```

### 0.5.2 Newlines in Single-quoted Strings

Note that there is no rule against having a single-quoted string span several lines. When you do this, the string has newline characters embedded in it.

A newline character is a special ASCII character that indicates that a new line should be started. In a text editor, or when printing output to the screen, this usually indicates that the cursor should move from the end of the current line to the first position on the line following it.

Since Perl permits the placement of these newline characters directly into single quoted strings, we are permitted to do the following:

```
, Time to
start anew . ;       # Represents the single string composed of:
    # ,Time to, followed by a newline, followed by
    # ,start anew ,
```

---

This string has a total of twenty characters. The first seven are Time to. The next character following that is a newline. Then, the eleven characters, start anew, follow. Note again that this is one string, with a newline as its eighth character.

Further, note that we are not permitted to put a comment in the middle of the string, even though we are usually allowed to place a # anywhere on the line and have the rest of the line be a comment. We cannot do this here, since we have yet to terminate our single-quoted string with a ‘, and thus, any # character and comment following it would actually become part of the single-quoted string! Remember that single-quotes strings are delimited by ‘ at the beginning, and ‘ at the end, and everything in between is considered part of the string, included newlines, # characters and anything else.

0.5.3 Examples of Invalid Single-quoted Strings

In finishing our discussion of single-quoted strings, consider these examples of strings that are not legal because they violate the exceptions we talked about above:

\'You cannot do this: \\; # INVALID: the ending \ cannot be alone
\'It is 5 o\,clock!, # INVALID: the , in o\,clock should be escaped
\'Three \\,s: \\\\,; # INVALID: the final \ escapes the ,,
\'Thus # INVALID: the literal is not terminated
\'This is my string; # INVALID: missing close quote

Sometimes, when you have invalid string literals such as in the example above, the error message that Perl gives is not particularly intuitive. However, when you see error messages such as:

(Might be a runaway multi-line string starting on line X)
Bareword found where operator expected
Bareword "foo" not allowed while "strict subs" in use

It is often an indication that you have runaway or invalid strings. Keep an eye out for these problems. Chances are, you will forget and violate one of the rules for single-quoted strings eventually, and then need to determine why you are unable to run your Perl program.

0.6 Brief Digression from Strings Alone: The print Function

Before we move on to our consideration of double-quoted strings, it is necessary to first consider a small digression. We know how to represent strings in Perl, but, as you may have noticed, the examples we have given thus far do not do anything interesting. If you try placing the statements that we listed as examples in Single Quoted Strings11, into a full Perl program, like this:

```
#!/usr/bin/perl
use strict;
```
use warnings;

'Three "\\s: "\\\\\\n"; # There are three \ chars between "
'xxx\'xxx'; # xxx, a single-quote character, and then
xxx
'Time to
start anew.';

you probably noticed that nothing of interest happens. Perl gladly runs this program, but it produces no output.

Thus, to begin to work with strings in Perl beyond simple hypothetical considerations, we need a way to have Perl display our strings for us. The canonical way of accomplishing this in Perl is to use the print function.

The print function in Perl can be used in a variety of ways. The simplest form is to use the statement print STRING;, where STRING is any valid Perl string.

So, to reconsider our examples, instead of simply listing the strings, we could instead print each one out:

#!/usr/bin/perl
use strict;
use warnings;

print 'Three "\\s: "\\\\\\n"'; # Print first string
print 'xxx\'xxx'; # Print the second
print 'Time to
start anew.
'; # Print last string, with a newline at the end

This program will produce output. When run, the output goes to what is called the standard output. This is usually the terminal, console or window in which you run the Perl program. In the case of the program above, the output to the standard output is as follows:

Three "\\s: "\\\\\\n"xxx'xxxTime to
start anew.

Note that a newline is required to break up the lines. Thus, you need to put a newline at the end of every valid string if you want your string to be the last thing on that line in the output.

Note that it is particularly important to put a newline on the end of the last string of your output. If you do not, often times, the command prompt for the command interpreter that you are using may run together with your last line of output, and this can be very disorienting. So, always remember to place a newline at the end of each line, particularly on your last line of output.

Finally, you may have noticed that formatting your code with newlines in the middle of single-quoted strings hurts readability. Since you are inside a single-quoted string, you cannot change the format of the continued lines within the print statement, nor put comments at the ends of those lines because that would insert data into your single-quoted strings. To handle newlines more elegantly, you should use double-quoted strings, which are the topic of the next section.
0.7 Double Quoted Strings

Double-quoted strings are another way of representing scalar string literals in Perl. Like single-quoted strings, you place a group of ASCII characters between two delimiters (in this case, our delimiter is "). However, something called interpolation happens when you use a double-quoted string.

0.7.1 Interpolation in Double-quoted Strings

Interpolation is a special process whereby certain special strings written in ASCII are replaced by something different. In Single-quoted strings section\textsuperscript{12}, we noted that certain sequences in single-quoted strings (namely, \" and \' ) were treated differently - these are called backslash escape sequences. This is very similar to what happens with interpolation.

For example, in interpolated double-quoted strings, various sequences preceded by a \ character act differently according to the chart below:

<table>
<thead>
<tr>
<th>String</th>
<th>Interpolated As</th>
</tr>
</thead>
<tbody>
<tr>
<td>\ \</td>
<td>an actual, single backslash character</td>
</tr>
<tr>
<td>$</td>
<td>a single $ character</td>
</tr>
<tr>
<td>@</td>
<td>a single @ character</td>
</tr>
<tr>
<td>&quot;</td>
<td>a single double-quote character</td>
</tr>
<tr>
<td>\t</td>
<td>tab</td>
</tr>
<tr>
<td>\n</td>
<td>newline</td>
</tr>
<tr>
<td>\r</td>
<td>hard return</td>
</tr>
<tr>
<td>\f</td>
<td>form feed</td>
</tr>
<tr>
<td>\b</td>
<td>backspace</td>
</tr>
<tr>
<td>\a</td>
<td>alarm (bell)</td>
</tr>
<tr>
<td>\e</td>
<td>escape</td>
</tr>
<tr>
<td>\056</td>
<td>character represented by octal value, 056 (same as .)</td>
</tr>
<tr>
<td>\x2E</td>
<td>character represented by hexadecimal value, 2E (same as .)</td>
</tr>
</tbody>
</table>

As you may have noticed in the previous chapter, you can put the name of a variable within a string with its leading dollar sign. This form of interpolation replaces the name of the variable in the string with the content of the variable.

0.7.2 Examples of Interpolation

Let us consider an example that uses a few of these characters:

```
#!/usr/bin/perl
use strict;
use warnings;
```
print "A backslash: \\
print "Tab follows:\tover here\nprint "Ring! \a\nprint "Please pay someone@example.org $20.\n"

This program, when run, produces the following output on the screen:

```
A backslash: \
Tab follows: over here
Ring!
Please pay someone@example.org $20.
```

In addition, when running, you should hear the computer beep. That is the output of the \a character, which you cannot see on the screen. However, you should be able to hear it.

Notice that the \n character ends a line. \n should always be used to end a line. Those students familiar with the C language will be used to using this sequence to mean newline. When writing Perl, the word newline and the \n character are roughly synonymous.

### 0.7.3 String Operators

*Operators* manipulate two or more strings in some way.

#### The Concatenation Operator

Perl uses the . operator to concatenate or connect two strings together, like this:

```
"Hello" . "World" # This is the same as "HelloWorld"
```

If you want to make the string have a space between Hello and World you could write it like this:

```
"Hello" . " " . "World" # This is the same as "Hello World"
```

Or like this:

```
"Hello" . "World" # This is the same as "Hello World"
```

#### The x Operator

This is called the *string repetition* operator and is used to repeat a string. All you have to do is put a string on the left side of the x and a number on the right side. Like this:
"Hello" x 5 # This is the same as "HelloHelloHelloHelloHello"

If you wish to insert a line break after each output of the string, use:

"Hello\n" x 5

0.7.4 Exercises

• Write a program that uses the . operator to print "Hello Sir!".
• Write another program which uses the x operator to print "HelloHelloHelloHello". Put comments in this program that explain how it works
• Remember to take some time to play with single and double quoted strings, the more practice you get, the better you will be.

0.8 Numbers

Numbers in Perl do not have to be enclosed in any kind of punctuation; they can be written as straight numbers.

0.8.1 Floating Point Numbers

Here are some acceptable floating point numbers:

0.1, -3.14, 2.71828...

0.8.2 Integers

Integers are all whole numbers and their negatives (and 0): {... -3, -2, -1, 0, 1, 2, 3 ...}.  
Here are a few examples of integers:

12, -50, 20, 185, -6654, 6654

The following examples are not integers:

15.5, -3.458, 3/2, 0.5
0.8.3 Non-decimal Numbers

I'll dwell on this topic for a little longer than the other types of numbers. In Perl you can specify not only decimal numbers, but also numbers in hex, octal, and binary. If you are not familiar with how these systems work, you can try these Wikipedia articles:

- Hexadecimal\(^{13}\)
- Octal\(^ {14}\)
- Binary\(^ {15}\)

In Perl you have to specify when you are going to write a non-decimal number. Binary numbers start with an 0b, so here are some possible binary numbers:

```
0b101011101
0b10
```

Octal numbers start with 0 ("zero"), so here are some possible octal numbers:

```
015462
062657
012
```

Hexadecimal numbers start with 0x, so here are some possible hexadecimal numbers:

```
0xF17A
0xFFFF
```

0.8.4 Number Operators

Just like strings, numbers have operators. These operators are quite obvious so I'll just give a quick example of each one.

**The +, -, /, and * Operators**

These operators are pretty obvious, but here are some examples:

```
100 + 1  # That's 101
100 - 1  # That's 99
100 / 2  # That's 50
100 * 2  # That's 200
```

\(^{13}\) http://en.wikipedia.org/wiki/Hexadecimal

\(^{14}\) http://en.wikipedia.org/wiki/Octal

\(^{15}\) http://en.wikipedia.org/wiki/Binary_numeral_system
Perl also has the familiar increment, decrement, plus-equals, and minus-equals operators from C:

```
$a++  # evaluate, then increment
++$a  # increment, then evaluate
$a--  # evaluate, then decrement
--$a  # decrement, then evaluate
$a += 5  # plus-equals operator, adds 5 to $a. Equivalent to $a = $a + 5
$a -= 2  # minus-equals operator, subtracts 2 from $a. Equivalent to $a = $a - 2
```

Now let’s look at one more operator that’s a little less obvious.

**The ** Operator**

The `**` operator is simply the exponentation operator. Here's another example:

```
2**4  # That's 16, same as 2^4
4**3**2  # that's 4**(3**2), or 4^9, or 262144
```

**Extra!**

The modulus operator (`%`) can be used to find the remainder when dividing two numbers. If that doesn't make sense now, that's fine, it's not that important. (Note, this returns 0 when used on floating point numbers)

### 0.8.5 Exercises

- Remember the `x` operator? Use a mathematical expression as the number of times to repeat the string, see what happens.
- Write a program like our original hello world program except make it print a mathematical expression.

In Perl, there are five types of variables: $calars, @rrays, %hashes, &subroutines, and *typeglobs.

### 0.9 Simple variables

Variables, called scalars, are identified with the `$` character, and can contain nearly any type of data. For example:

```
$my_variable = 3;  # integers
$my_variable = 3.1415926;  # floating point
$my_variable = 3.402823669209384634633e+38;  # exponents
$my_variable = $another_variable + 1;  # mathematical operation
$my_variable = ,Can contain text,;  # strings
$my_variable = \$another_variable;  # scalar reference
$my_variable = @{array_variable};  # array reference
```

```
0.9.1 Case sensitivity

Note that the perl interpreter is case sensitive\textsuperscript{16}. This means that identifier names containing lowercase letters will be treated as being different and separate from those containing uppercase letters.

0.10 Arrays

Arrays in Perl use the @ character to identify themselves.

```perl
@my_array = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  # numeric list
@my_array = (1 .. 10);                      # same as above
@my_array = qw/John,, Paul,, Kanai,, Mahenge/; # strings
@my_array = qw/John Paul Kanai Mahenge/;      # the same - one-word strings, with \texttt{less} typing
@my_array = qw/red blue 1 green 5/;           # mixed types
@my_array = (@Array1, @Array2, @Array3);     # array of arrays
```

```perl
foreach my $Item (@my_array) {
  print "Next item is $Item \n";
}
```

However, when you deal with just one element of the array (using square brackets so it’s not confused), then that element of the array is considered a scalar which takes the $ sigil:

```perl
@my_array[0] = 1;
```

As in the C programming language, the number of the first element is 0 (although as with all things in Perl, it’s possible to change this if you want). Array subscripts can also use variables:

```perl
@my_array[@MyNumber] = 1;
```

0.11 Associative arrays

Associative arrays, or "hashes," use the % character to identify themselves.

```perl
%my_hash = {key1 => value1, key2 => value2};
```

When using the => the left side is assumed to be quoted. For long lists, lining up keys and values aids readability.

```perl
%my_hash = {
  key1 => value1,
  key2 => value2,
  key3 => value3,
};
```

However, when you deal with just one element of the array (using braces), then that element of the array is considered a scalar and takes the $ identifier:

\textsuperscript{16} http://en.wikibooks.org/wiki/%2Fcase%20sensitivity%2F
$my_hash{key1} = 'value1';

Associative arrays are useful when you want to refer to the items by their names.

## 0.12 Subroutines

Subroutines are defined by the `sub` function, and used to be called using `&` (using `&` is now deprecated). Here's an example program that calculates the Fibonacci sequence:

```perl
sub fib {
    my $n = shift;
    return $n if $n < 2;
    return fib( $n - 1 ) + fib( $n - 2 );
}
print fib(14);
```

The `if` statement is the primary conditional structure in Perl. The syntax is as follows:

```perl
if (boolean expression) {
    statement;
}
```

If the `boolean expression` evaluates to true, the statements between the two braces will be executed. The braces around statements are mandatory, even if there is only one statement (unlike C or Java).

An alternative syntax to the `if` statement may be used on a single statement. This involves putting the conditional at the end of the statement rather than before, and does not include braces:

```perl
statement if (boolean expression);
```

The following statements are synonymous:

```perl
if ( $x == 20 ) { print "hello";
}
print "hello" if ( $x == 20 );
```

You should choose whichever one is clearer in a given situation. For example, the following is legal, but unclear:

```perl
foreach my $word (@words) {
    if ($word eq 'end') { last; }
    print "$word\n";
}
```

This hides the `last` (which is like `break`, and ends the loop) over at the right. Instead use a postfix `if`:

```perl
foreach my $word (@words) {
    last if $word eq 'end';
    print "$word\n";
}
```

The `boolean expression` conditional can contain any one of the comparison operators covered in the next section.
Multiple conditions can be checked together using the boolean expression operators:

- `&&` - logical and, C style; used for most conditionals
- `and` - logical and, but with a lower precedence; used for flow control
- `||` - logical or, C style; used for most conditionals
- `or` - logical or, but with a lower precedence; used for flow control
- `!` - logical not, C style
- `not` - logical not, but with a lower precedence

```c
if ( ($x == 20) || ( ($x > 0) && ($x < 10) && !($x == 5) ) ) {
    print "x is equal to 20 or either between 0 and 10 but not 5.\n"
}
```

Conditional statements can also be extended with the `elsif` and `else` structures:

```c
if ( , , boolean expression 1 , , ) {
    , , statement 1 , ,
}
elsif ( , , boolean expression 2 , , ) {
    , , statement 2 , ,
}
else {
    , , statement 3 , ,
}
```

Note that an `if` statement is followed by any number (including zero) of `elsif` statements, and finally an optional `else` statement. The statements of an `elsif` will be executed if its boolean expression is true, and no preceding `(els)` `if` statement's boolean expression is true. The trailing `else` (if present) is executed if none of the preceding statements' boolean expressions are true.
1 Introduction

Perl’s set of operators borrows extensively from the C programming language\(^1\). Perl expands on this by infusing new operators for string functions (\(=\), \(\times\), \texttt{eq}, \texttt{ne}, etc.). C by contrast delegates its subset of Perl functionality to a library \texttt{strings.h}, and \texttt{ctype.h}, and includes no such functionality by default compilation. Perl also includes a highly flexible Regex\(^2\) engine inspired by Sed with improvements to standard POSIX regexes, most notably the support of Unicode.

\(^1\) \url{http://en.wikibooks.org/wiki/C%20programming%20language}
\(^2\) Chapter 4.24 on page 75
2 The operators

2.1 Arithmetic

Most arithmetic operators are binary operators; this means they take two arguments. Unary operators only take one argument. Arithmetic operators are very simple and often transparent.

2.1.1 Binary

All the basic arithmetic operators are present: addition (+), subtraction (–), multiplication (×), and division (/).

The modulus operator is %. Modulus returns the remainder of a division (/) operation.

```plaintext
# 3 goes into 4, 1 time with 1 left over.
print 4 % 3; # prints 1

# 2 goes into 4, 2 times with 0 left over.
print 4 % 2; # prints 0

# 3 goes into -4, -2 times with 2 left over.
print -4 % 3; # prints 2
```

The exponentiation operator is **. It allows you to raise one value to the power of another. If you raise to a fraction you will get the root of the number. In this example the second result when raised to the power of 2 should return \( 2 \) (\( 2^{1/2} \times 2 = 2 \)).

```plaintext
# Four squared:
print 4 ** 2; # prints 16

# Square root of 2
print 2 ** (1/2); # prints 1.4142135623731
```

The function sqrt is provided for finding a Square Root. Other fractional powers (i.e., (1/5), (2/13), (7/5), and similar) are suitably found using the ** operator.

2.1.2 Unary

The auto-decrement (––), and auto-increment (++) operators are unary operators. They alter the scalar variable they operate on by one logical unit. On numbers, they add or subtract one. On letters and strings, only the auto-increment shift one up in the alphabet, with the added ability to roll-over. Operators that come in post- and pre- varieties can be used two ways. The first way returns the value of the variable before it was altered, and the second way returns the value of the variable after it was altered.
The operators

```perl
my $foo = 1;
# post decrement (printed and then decremented to 0)
print $foo--; # prints 1
print $foo;  # prints 0

my $foo = 1;
# pre-decrement (decremented to 0 then printed)
print --$foo; # prints 0
print $foo;  # prints 0

my $foo = ,d,;
# pre-increment (incremented to e then printed)
print ++$foo; # prints e
print $foo;  # prints e

my $foo = ,Z,;
# post-increment (printed the incremented to AA)
print $foo++; # prints 2
print $foo;  # prints AA
```

2.2 Assignment

The basic assignment operator is "=" which sets the value on the left side to be equal to the value on the right side. It also returns the value. Thus you can do things like $a = 5 + ($b = 6), which will set $b to a value of 6 and $a to a value of 11 (5 + 6). Why you would want to do this is another question.

The assignment update operators from C, "+=", "-=", etc. work in perl. Perl expands on this basic idea to encompass most of the binary operators in perl.

<table>
<thead>
<tr>
<th>operator</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>+=</td>
<td>add assign, plus-equals</td>
</tr>
<tr>
<td>-=</td>
<td>subtract assign, minus-equals</td>
</tr>
<tr>
<td>*=</td>
<td>multiply assign</td>
</tr>
<tr>
<td>/=</td>
<td>divide assign</td>
</tr>
<tr>
<td>%=</td>
<td>modulo assign</td>
</tr>
<tr>
<td>**=</td>
<td>exponent assign</td>
</tr>
<tr>
<td>.=</td>
<td>concatenate assign</td>
</tr>
<tr>
<td>&amp;&amp;=</td>
<td>logical AND assign</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;=</td>
<td>bitwise AND assign</td>
</tr>
<tr>
<td></td>
<td>=</td>
</tr>
<tr>
<td>^=</td>
<td>bitwise XOR assign</td>
</tr>
<tr>
<td>-=</td>
<td>bitwise NOT assign</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>left shift assign</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>right shift assign</td>
</tr>
</tbody>
</table>
2.3 Comparison

Perl uses different operators to compare numbers and strings. This is done because in most cases, Perl will happily *stringify* numbers and *numify* strings. In most cases this helps, and is consistent with Perl's *DWIM* Do-What-I-Mean theme. Unfortunately, one place this often does not help, is comparison.

<table>
<thead>
<tr>
<th>name</th>
<th>numeric</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>equal</td>
<td>==</td>
<td>eq</td>
</tr>
<tr>
<td>not equal</td>
<td>!=</td>
<td>ne</td>
</tr>
<tr>
<td>less than</td>
<td>&lt;</td>
<td>lt</td>
</tr>
<tr>
<td>greater than</td>
<td>&gt;</td>
<td>gt</td>
</tr>
<tr>
<td>less or equal</td>
<td>&lt;=</td>
<td>le</td>
</tr>
<tr>
<td>greater or equal</td>
<td>&gt;=</td>
<td>ge</td>
</tr>
<tr>
<td>compare</td>
<td>&lt;=&gt;</td>
<td>cmp</td>
</tr>
</tbody>
</table>

2.4 Logical

Perl has two sets of logical operators, just like the comparison operators, however not for the same reason.

The first set (sometimes referred to as the C-style logical operators because they are borrowed from C) is &&, ||, and !. They mean logical AND, OR, and NOT respectively. The second set is and, or, and not.

The only difference between these two sets is the precedence they take (See Precedence\(^1\)). The symbolic operators take a much higher precedence than the textual.

2.4.1 Conditionals

Most of the time, you will be using logical operators in conditionals.

```perl
# Only prints "I like cookies
" if both $a is 5 and $b is 2
if($a == 5 && $b == 2){
  print "I like cookies
";
}
```

\(^1\) Chapter 2.10 on page 24
The operators

In this case, you could safely substitute `and` for `&&` and the conditional would still work as expected, however, this is not always the case.

```plaintext
#True if $a$ is 5, and either $b$, $c$, or both are 2
if($a == 5 and $b == 2 || $c == 2){
    print "I like cookies\n";
}
#Using brackets, the order is made more clear.
#This conditional acts in the same way as the last.
if(($a == 5 and ($b == 2 || $c == 3)){
    print "I like cookies\n";
}
```

This, however, is completely different.

```plaintext
if($a == 5 && $b == 2 or $c == 3){
    print "I like cookies\n";
}
#Equivalent and easier to understand with brackets
if(($a == 5 && $b == 2) or $c == 3}{
    print "I like cookies\n";
}
```

Most people prefer to use C-style logical operators and use brackets to enforce clarity rather than using a combination of textual and C-style operators (when possible), which can be very confusing at times.

### 2.4.2 Partial evaluation

Partial evaluation (or "short circuiting") is the property of logical operators that the second expression is only evaluated if it needs to be.

```plaintext
($a, $b) = (5, 2);
#$b < 3 is not evaluated at all because when the interpreter
#finds that $a == 4 is false, there is no need to evaluate $b < 3
#because the conditional is automatically false
if($a == 4 && $b < 3){
    print "I like cookies\n";
}
```

This also works with logical OR statements. If the first expression evaluates as true, then the second is never evaluated because the conditional is automatically true.

This becomes useful in a case like this:

```plaintext
sub foo {
    #returns a true or false value
}
foo() or print "foo() failed\n";
```

Here, if the `foo()` subroutine returns false, then "foo() failed\n" is printed. However, if it returns true, then "foo() failed\n" is not printed, because the second expression (print "foo() failed\n") does not need to be evaluated.
2.5 Bitwise

These operators perform the same operation as the logical operators, but instead of being performed on the true/false value of the entire expressions, it is done on the individual respective bits of their values.

- "&" (bitwise AND)
- "|" (bitwise OR)
- "^" (bitwise XOR)
- "~" (bitwise NOT)

The left and right shift operators move the bits of the left operand (e.g. $a$ in the case of $a \ll b$) left or right a number of times equal to the right operand ($b$). Each move to the right or left effectively halves or doubles the number, except where bits are shifted off the left or right sides. For example, $\text{number} \ll 3$ returns $\text{number}$ multiplied by 8 ($2^3$).

- "\ll" (left shift)
- "\gg" (right shift)

2.6 String

The string concatenation operator is ., not + which some other languages use.

```perl
print 'Hello, ., world,'; # prints "Hello world" without a newline at the end

my $str = "hi";
my $repeated_str = $str x 5;
print "$repeated_str\n"; # prints "hihihihihi" with a newline at the end
```

2.7 Comparing strings

To compare strings, use eq and ne instead of == or != respectively. You can also look for a substring with substr(), or pattern-match with regular expressions.

2.8 File Test

See Perl Programming/Function Reference#-X²
2.9 Other

The range operator (..) returns a list of items in the range between two items; the items can be characters or numbers. The type of character is determined by the first operand; the code:

```perl
print ('A'..'Z');
print ('a'..'z');
print ('A'..'z');
print (1..'a');
print (1..20);
print ('&'..'!');
print (10..-10);
print "$_\n" foreach (1..10);
```

Outputs (Newlines added for readability):

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891011121314151617181920
&
1
2
3
4
5
6
7
8
9
10
```

Note that the case is defined by the first operand, and that the 1..'a' and (10..-10) operations don't return anything.

2.10 Precedence

Precedence is a concept that will be familiar to anyone who has studied algebra or coded in C/C++. Each operator has its place in a hierarchy of operators, and are executed in order. The precedence of perl operators is strict and should be overridden with parentheses, both when you are knowingly going against precedence and when you aren't sure of the order of precedence. For a complete listing of the order, check `perlop`.

2.11 The smart match operator (˜˜)

The **smart match operator** is new in Perl 5.10. To use it, you'll need to explicitly say that you're writing code for Perl 5.10 or newer:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.10.0;  # We will be using the smart match operator

my $scalar = 'hi';
my @array = qw(one two three);
my %hash = (
    hi => 1,
    ho => 2,
    he => 3,
);

if ($scalar ~~ @array) { print "1\n"; }  # Doesn't print; 'hi' isn't an element in @array
if ($scalar ~~ %hash) { print "2\n"; }  # Does print; 'hi' is a key in %hash
if (@array ~~ %hash) { print "3\n"; }  # Doesn't print; none of the elements of @array match a key in %hash
```

The smart match operator is versatile and fast (often faster than the equivalent comparison without ˜˜). See smart matching in detail\(^4\) for the comparisons it can do. ˜˜ is also used in the `given/when` switch statement new in 5.10, which will be covered elsewhere.

2.12 Dereferencing

2.12.1 The doubledollar

A variable, previously referenced with the reference operator can be dereferenced by using a doubledollar symbol prefix:

```
$number = 12;
$refnum = $number;  # backslash is the reference operator
$$refnum = 13;  # $$ is used as a dereference to the original variable
print $$refnum;  # This is an alternative syntax using brackets
```

2.12.2 The arrow operator

If the left hand operand of the arrow operator is an array or hash reference, or a subroutine that produces one, the arrow operator produces a look up of the element or hash:

```
$result = $hashreference->{$key};  # look up a hash key from a reference variable
@arrayslice = $arrayreference->[3 .. 5];  # obtain a slice from an array reference
```
Perl has four fundamental data types: scalars, lists, hashes, and typeglobs.

**scalar**

is a funny way of saying a single value; it may be a number, a string, or a reference.

**list**

is an ordered collection of scalars. A variable that holds a list is called an *array*. Items in a list or array can be accessed by their position in the list; programs can retrieve the first, second, third, etc. item in a list.

**hash**

is like an array, in that a hash holds many values, but the values are identified by a unique "key", rather than an ordinal index position.

**typeglob**

is a variable representing an entry within the internal symbol table. It is used to manipulate file handles, and to create references or aliases.

All variables are marked by a leading sigil[^7], which identifies the data type. The same name may be used for variables of different types, without conflict.

```perl
$foo   # a scalar
@foo   # a list
%foo   # a hash
*foo   # a typeglob
```

## 2.13 Scalar Variables

### 2.13.1 Introduction to Scalar Variables

Now that you understand how to use strings and numbers in Perl, you need to start learning how to use variables. The best way to learn about scalar variables - Perl talk for a single variable, as against a group or list of values - is to look at an example.

```perl
#!/usr/bin/perl
use warnings;
$my_scalar_variable = "Hello, Sir!\n"
print $my_scalar_variable;
```

Now let's break this program down:

- The first two lines you already know, `#!/usr/bin/perl` and `use warnings;`
- The third line is more interesting, it contains a scalar variable. There are a few important things to point out:

1. In case you haven't figured this out, the scalar variable in this line is \$my_scalar_variable

2. Notice the \$ before the name my_scalar_variable, in order to define a scalar variable, this sign must appear before the name.

• Now let's look at the last line. This is just the familiar print function being told to print the value of \$my_scalar_variable.

Try it!
Type in the program mentioned above and run it.

2.13.2 Assigning and Using Scalar Variables

In the course of writing a program, you will most likely use a variable. What is a variable? A variable is something that stores data. A scalar variable holds a single value.

Naming Conventions

• All scalar variables names must start with a \$ symbol. You can remember this by thinking $calar.
• Variable names can be comprised of alphanumeric characters and underscores.
• Numeric characters are allowed in names of variables, but not as the first character after the \$.

Using Scalar Variables

Scalar Variables and Strings

You may recall that earlier in the book, I said that whether you use " or ’ in strings makes a big difference in the interaction of strings and variables. Well now I am going to explain what I meant.

Now that you know what a variable is, what if you wanted to put a variable in a string? Here's the difference:

• With a double quoted string, this program:

```perl
#!/usr/bin/perl
use warnings;
$variable = 4;
print "I saw $variable lions!";
```

Would return "I saw 4 lions!"

• With a single quoted string, this program:

```perl
#!/usr/bin/perl
use warnings;
$variable = 4;
```
print "I saw $variable lions!;"

Would return "I saw $variable lions!"

Try it!
Type in the programs mentioned above and run them.

This effect is because of what I said before, single quoted strings\(^8\) are interpreted literally.

### 2.13.3 Comparison Operators

*Main article: Perl Programming/Operators\(^9\)*

There are operators that are used for comparing numbers and strings. This can be very useful when you get to more advanced programming. Both numbers and strings have their own set of operators which test for a condition such as equal or not equal and return either true or false.

#### Numeric Comparison Operators

Here is the list of numeric comparison operators:

- `==` - Equal to
- `!=` - Not equal to
- `<` - Less than
- `>` - Greater than
- `<=` - Less than or equal to
- `>=` - Greater than or equal to
- `<>` - Numeric Comparison

#### String Comparison Operators

Here is the list of string comparison operators:

- `eq` - Equal to
- `ne` - Not equal to
- `lt` - Less than
- `gt` - Greater than
- `le` - Less than or equal to
- `ge` - Greater than or equal to
- `cmp` - String Comparison

---

\(^8\) Chapter 0.5 on page 5
\(^9\) Chapter 0.12 on page 16

28
Note
The two 'Comparison' operators <=> and cmp are slightly different from the rest. Rather
than returning only true or false, these operators return 1 if the left argument is greater than
the right argument, 0 if they are equal, and -1 if the right argument is greater than the left
argument.

2.13.4 Exercises

• Try writing a program like the Hello World program except elaborate it by storing "Hello,
  world!\n" in a variable and then printing the variable.

• Play around with all the things we have learned so far. Try to create a program that has an example
  of everything we have learned so far.

Perl syntax includes both lists and arrays.

2.14 Lists

A list in perl is an ordered set of scalar values. It is represented in your code as a comma-separated
sequence of values, which may or may not be contained in scalar variables. Lists can be used to make
multiple assignments at once, and can be passed as arguments to several built-in and user-defined
functions:

#!/usr/bin/perl
use strict;
use warnings;
my ($length, $width, $depth) = (10, 20, 15);
print "The values are: ", $length, $width, $depth;

Note
Parentheses are not required in the construction of a list. They are used only for precedence.

2.14.1 Alternate List Construction

When creating a list of several strings that do not include spaces, Perl provides a shortcut to get
around typing multiple quotes and commas. Instead of

($name1, $name2, $name3, $name4) = ("Paul", "Michael", "Jessica", "Megan");

you can use the qw// operator. This operator uses any non-alpha-numeric character as a delimiter
(typically the / character), and encloses a space-separated sequence of barewords. A delimiter
separates the command with the arguments. The above line is identical to the following:

($name1, $name2, $name3, $name4) = qw/Paul Michael Jessica Megan/;

and both are equal to this:
The operators

($name1, $name2, $name3, $name4) = qw(Paul Michael Jessica Megan);

The last example uses the open and close parenthesis as a different delimiter. If there is an open and close version of the delimiter you choose, you need to use them both. Otherwise just repeat the same symbol twice. For example, you cannot type qw<Paul Michael> you have to type qw<Paul Michael>.

You can also abuse the glob syntax, when the strings do not include shell metacharacters:

($name1, $name2, $name3, $name4) = <Paul Michael Jessica Megan>;

Note
The resulting strings from the qw// operator are single-quoted, meaning no interpolation happens in the set. If you need to include a variable in your list, you cannot use this method.

2.14.2 List Assignments

As shown above, lists can be used to make several assignments at once. If the number of variables on the left is the same as the number of values on the right, all variables are assigned to their corresponding values, as expected.

If there are fewer variables on the left than values on the right, the 'extra' values are simply ignored:

#!/usr/bin/perl
($length, $width) = (10, $w, 15); # $length gets 10, $width gets the value of $w. 15 is ignored

If there are more variables on the left than values on the right, the 'extra' variables are assigned the default undef value:

#!/usr/bin/perl
($length, $width, $depth) = (10, $w); # $length gets 10, $width gets the value of $w. $depth is undef

The existence of list assignment creates the ability to 'swap' two variables' values without the need of an intermediary temporary variable:

#!/usr/bin/perl
$foo = 10;
$bar = 5;
($foo, $bar) = ($bar, $foo); # $foo now equals 5, while $bar equals 10;
2.15 Arrays

An array in Perl is a variable which contains a list. An array can be modified, have elements added
and removed, emptied, or reassigned to an entirely different list. Just as all scalar variables start with
the $ character, all array variables start with the @ character.

Note
It is a common and frequent mistake in Perl to use the terms 'list' and 'array' interchangeably.
They do not have the same meaning.
A decent analogy is that a list (such as qw/foo bar baz/) is to an array (such as
@values) as a string (such as 'Paul') is to a scalar variable (such as $name).

2.15.1 Array Assignment

Arrays are assigned lists of values. The list of values can be arbitrarily large or small (it can even
contain 0 elements).

#!/usr/bin/perl
@nums = (1,2,3,4,5);
@more = 6..1000; #using the ,range, operator
@none = (); # empty array.
@names = qw/Paul Michael Jessica Megan/;
@all = (@nums, @more); #@all contains all integers from 1 to 1000

That last example exemplifies a feature of Perl known as 'array flattening'. When an array is used in a
list, it is the array's elements that populate the list, not the array itself. As stated above, a list is a set
of scalar values only. Therefore, the @all array contains 1000 elements, not 2.

Note
Although this implies you cannot create an 'array of arrays', or 'two-dimensional arrays', such
things do exist in Perl. They are simulated by using references.

2.15.2 Arrays in Scalar context

When an array is used in scalar context - either by assigning a scalar variable to the array's value, or
using it in an operation or function that expects a scalar - the array returns its size. That is, it returns
the number of elements it currently contains

#!/usr/bin/perl
@names = (Paul,,,Michael,,,Jessica,,,Megan,);
$how_many = @names;

10 Chapter 4.21 on page 75
print "I have a total of $how_many names\n";

Note
A common misconception is that a list in scalar context will also return its size. This is untrue. In fact, there is no such thing as a list in scalar context: using the comma operator in a scalar context does not create a list, instead it evaluates each of its arguments, left to right, and returns the last one:

```
$name = ('Paul','Michael','Jessica','Megan');
print "The last name in my list is $name\n";
```

2.15.3 Printing an Array

There are two general ways of printing the values of an array. You can either print the list of items in the array directly, or you can interpolate the array in a double-quoted string.

```
@names = qw/Paul Michael Jessica Megan/;
print "My names are: @names .\n";
print "My names are: @names .\n";
```

In the first example, the print function is being given a list of 6 arguments: the string 'My names are: ', each of the four values in @names, and the string "\n". Each argument is printed separated by the value of the $, variable (which defaults to the empty string), resulting in the values from the array being 'squished' together:

My names are: PaulMichaelJessicaMegan.

In the second example, the print function is being given exactly one argument: a string that contains an interpolated array. When Perl interpolates an array, the result is a string consisting of all values in the array separated by the value of the $" variable (which defaults to a single space):

My names are: Paul Michael Jessica Megan.

Note
Both the $, and $" variables can be changed to any string you like. For example, to separate the array's items with a comma and a space instead of just a space:

```
$" = ', ';
print "My names are: @names .\n";
```

My names are: Paul, Michael, Jessica, Megan.

You generally do not want to do that as this may cause problems in other parts of your program depending on the default values of those variables though! A safer way to print your arrays with custom separator will be explained later.
2.15.4 Accessing Elements of an Array

The elements of an array are accessed using a numerical reference within square brackets. Because each item within an array is a scalar value, you need to use $ when referencing a value. The first element of an array is number 0 and all the others count up from there.

A negative number will count down from the right side of the array. This means that -1 references the last element of the array and -3 references the third to last element. Let's see some examples:

```perl
@array = (1, 2, 3, 4, 5);
print $array[0];  # Prints 1
print $array[3];  # Prints 4
print $array[-1]; # Prints 5
```

What if you need to know the last index? $#array will return it for you:

```perl
@array = (1, 2, 3, 4, 5);
print $array[4];  # Prints 5
print $array[-1];  # Same as above
print $array[ $#array ]; # Also prints 5
```

A common mistake is to do this:

```perl
print @array[0];  # Also prints 1, but for the wrong reasons
```

In fact @array[0] is a slice (that is, a sub-array of an array) that contains one element, whereas $array[0] is a scalar which contains the value 1.

2.15.5 Common Array Functions

2.16 Command line arguments

As you may wonder, Perl scripts support command line arguments. The entire list of parameters is stored in the array @ARGV, with the first entry containing the first command line argument. If no command line parameters were passed, @ARGV is an empty array.

The array functions and operators listed above can easily be used to detect the passed command line arguments and to detect the number of arguments provided.

2.17 Related Articles

- Data Structures/Arrays\(^\text{11}\)
- List Functions\(^\text{12}\)
- Array Functions\(^\text{13}\)


\(^{12}\) Chapter 4.49 on page 104

\(^{13}\) Chapter 4.48 on page 103
The operators

- Perl Arrays

A Perl hash is similar to an ordinary array, but instead of using integer indexes, a hash uses "keys" that can take on any scalar value. These are usually strings or numbers.

Syntax: instead of the @ operator, associative arrays use the % symbol, and rather than square brackets [], as in $myarray[0], hash elements are referenced using curly brackets {}, as in $myhash{"george"}.

Hashes are one of the most powerful and commonly used features in Perl. A typical use would be to build a hash that contains a "dictionary", with each key being a word in the dictionary, and the corresponding values being the definitions of those words.

A hash containing the sounds various household pets make is below

```perl
my %petsounds = ("cat" => "meow",
               "dog" => "woof",
               "snake" => "hiss");

'=' and ',' are actually interchangeable, so the right side could look exactly like an array. This means that you can assign an array to a hash. In such an assignment, each element with an even index (starting from 0) in the array becomes a key in the hash. The following statements create the same hash as the previous one does

```perl
my @array = ("cat", "meow", "dog", "woof", "snake", "hiss");
my %petsounds = @array;
```

But the first style is more preferred because it makes the statement more readable.

To access a hash element, use the curly brackets:

```perl
print STDOUT "The cat goes " . $petsounds{"cat"} . ".\n";
```

will print the following to STDOUT

```
The cat goes meow.
```

To add a new sound item to a hash

```perl
$petsounds{"mouse"} = "squeak!";
```

To overwrite an existing element, just reassign it

```perl
$petsounds{"dog"} = "arf!"; # The dog now goes "arf!"
```

To remove an item from a hash, use delete. Setting the value to undef does not delete the item; using exists on a key that has been set to undef will still return true.

```perl
delete($petsounds{"cat"}); # will remove "cat" from our hash
```

---

14  http://www.programmingbulls.com/perl-array
2.18 "Associative Arrays"

Originally, a "hash" was called an "associative array", but this term is a bit outdated (people just got sick and tired of using seven syllables). Although it isn’t intuitive for newcomers to programming, "hash" is now the preferred term. The name is derived from the computer science term, hashtable.\(^{15}\)

2.19 Working with hashes

2.19.1 Printing hash contents

If you know PHP, you may have thought by now of some convenient way to print the contents of your array the way `print_r` does...

```
use Data::Dumper;
print Dumper(\%hash);
```

2.19.2 Counting the number of entries in a hash

To get the size of the hash, simply find the size of the result of the `keys` function, by evaluating it in scalar context:

```
my %hash = {
    ,key1, => 1,
    ,key2, => 2
};
print "Hash has ", scalar(keys(%hash));
```

2.20 Hash of Hashes of Hashes

You can define multidimensional hash array variables. An example may look like this:

```
#!/usr/bin/perl
use Data::Dumper;

my %a = ();
$a{1}{"a"}="FIRST";
$a{1}{"c"}="THIRD";
$a{1}{"b"}="SECOND";
foreach my $k1 ( sort keys %a ) {
    foreach my $k2 ( sort keys %{$a{$k1}} ) {
        foreach my $k3 ( sort keys %{$a{$k1}{$k2}} ) {
            print "$k1\t$k2\t$k3\t$a{$k1}{$k2}{$k3}\n";
        }
    }
}
```

The operators

} }
}

print Dumper(\%a);

This code will produce:

```
1 a A FIRST
1 b C SECOND
1 c B THIRD
$VAR1 = {
  '1' => {
    'c' => {
      'B' => 'THIRD'
    },
    'a' => {
      'A' => 'FIRST'
    },
    'b' => {
      'C' => 'SECOND'
    }
  }
};
```

Input/Output or **IO**, is an all-encompassing term that describes the way your program interacts with the user. IO comes in two forms, or *stream* types: the program's stimuli are collectively referred to as *input*, while the medium that the program uses to communicate back, write logs, play sounds, etc. is known as *output*. Both types of streams can be redirected either at a lower level than Perl, as is the case when done through the operating system by the shell; or, in Perl itself, as is the case when you reopen the file handles associated with the stream.

### 2.21 Output

You have already learned how to output with the `print` statement. A simple reference is provided:

```
print "Hello World";
```

What this `print` statement is actually doing is printing to *STDOUT*, which stands for *standard output*. Standard output is the default destination for all output. If you wish to print anywhere else you must be explicit. We will revisit this later.

### 2.22 Input

As you may have imagined, it's very hard to write a good program without any type of input; here is an example program to teach you these concepts:

```
#!/usr/bin/perl
use strict;
use warnings;

print "What is your name?\n";
```
Input

```perl
## Get the users $name from Standard In
my $name = <STDIN>;

print "Your name is $name
";
```

Standard input is usually the keyboard though this can be changed at a lower level than your program. For now we will assume it isn’t changed. However, this might not be an assumption you wish to make in production code.
3 Unit Exercise

- Write a program which prompts the user for a number and then returns the number multiplied by four (or any other number).

3.1 Advanced Output Overview

In many situations, especially for web programming, you will find that you want to put certain things, such as backslashes or quotes, in your text that aren’t allowed in a traditional print statements. A statement such as

```perl
print "I said "I like mangos and bananas".
```

will not work because the interpreter would think that the quotes mark the end of the string. As with all things in Perl, there are many solutions to this problem.

3.2 Use other quotes

The quickest solution to this problem would be to use single quotes to surround the string, allowing the use of double quotes in the middle.

```perl
# I said "I like mangos and bananas".
print 'I said "I like mangos and bananas".';
```

This is obviously not the best solution, as it is conceivable that you are trying to print a string containing both kinds of quote:

```perl
# I said "They're the most delicious fruits".
print 'I said "They're the most delicious fruits".';
```

3.3 Escape characters

For situations like the above where only a short amount of text is being quoted, a common solution is to escape any quotes in the string. By preceding any quotes with a backslash they are treated as literal characters.
Using single quotes, the characters that require escaping are \'.

Using double quotes, the characters that need escaping are the variable sigils, (i.e. $@%*) in addition to "

Using \ to escape reserved characters of course implies that you also need to escape any backslashes you want to use in your string. To print the second line literally using perl, you would need to write:

```perl
print "I said \"They\'re the most delicious fruits\".
```

Luckily perl provides us with another way of quoting strings that avoids this problem.

### 3.4 Custom Quotes

Perl provides the operators `q` and `qq` that allows you to decide which characters are used to quote strings. Most punctuation characters can be used. Here are a few examples:

```perl
print q{ I said "They\'re the most delicious fruits!". }
print qq{ I said "They\'re the most delicious fruits\"." };
```

The only symbols I have found that cannot be used for these quotes are $ ' /

### 3.5 Block Output

As can be seen, while the custom quotes option works for short strings, it can run into problems if a lot of text containing a lot of punctuation is output. For this situation, a technique called Block quoting can be used.

```perl
print <<OUTPUT
 I said "They\'re the most delicious fruits!".
OUTPUT
```

Any string of characters can be used instead of OUTPUT in the example above. Using this technique anything can be output no matter what characters it contains. The one caveat of this method is that the closing OUTPUT must be the first character on the line, there cannot be any space before it.

```perl
print <<EverythingBetween
 ... ...
EverythingBetween
```

### 3.6 Variable Output

It is possible to output variables within strings when you use some of these methods:
my $one = 'mangoes';

print "I like $one.";  # I like mangoes.
print $one;  # I like $one.
print qq$one;  # I love $one.
print q$one;  # I love $one.

print <<<OUT
I love $one
OUT;  # I love mangoes

print <<<OUT
I love $one.
OUT;  # I love $one.

Perl will figure out where your variable ends if the character after it is neither a letter, number nor an underscore. If that is not your case, put your variable inside curly braces:

my $one = 'lemon';

print "A $one is too sour;

";  # A lemon is too sour;
print "$oneade is better.

";  # lemonade is better.

print <<<OUT
I love ${one}s in $one souffle.

OUT;  # I love lemons in lemon souffle.

3.7 Caveats

The single quote ' q{ and double quote " qq <<A operators, behave differently. Whereas when using double quotes, you can include variables and escape any characters, when you use single quotes you can only escape single quotes and you cannot include variables.

3.8 Control structures

The basic control structures do not differ greatly from those used in the C¹ programming language or Java² programming language:

3.8.1 Loops

while ($boolean) {
  # do something
}

until ($boolean) {

¹ http://en.wikibooks.org/wiki/Programming%3AC
² http://en.wikibooks.org/wiki/Programming%3AJava
Note that the statements in a while (or until) loop are not executed if the Boolean expression evaluates to false (or true, respectively) on the first pass.

```perl
do {
    # something
} while ($boolean);
```

The `do {} while` and `do {} until` loops are technically statement modifiers and not actual control structures. The statements will be executed at least once.

```perl
for (my $i=0 ; $i<10 ; $i++) { # for (initialization; termination condition; incrementing expr) { ... }
    print "$i\n";
}
```

```perl
foreach my $variable (@list) {
    print "$variable\n";
}
```

$variable is an alias to each element of the @list, starting at the first element on the first pass through the loop. The loop is exited when all the elements in the list have been exhausted. Since $variable is an alias, changing the value will change the value of the element in the list. This should generally be avoided to enhance maintainability of the code.

If $variable is omitted, the default variable $_ will be used.

Note that for and foreach are actually synonyms and can be used interchangeably.

### 3.8.2 If-then statements

```perl
if ($boolean_expression) {
    # do something
}
```

```perl
unless ($boolean_expression) {
    # do something
}
```

Statements with else blocks (these also work with unless instead of if)

```perl
if ($boolean) {
    # do something
} else {
    # do something else
}
```

```perl
if ($boolean) {
    # do something
}
```

---

3 Chapter 4.2 on page 46
elsif ($boolean) {
    # do something else
}

Control statements can also be written with the conditional following the statements (called "postfix"). This syntax functions (nearly) identically to the ones given above.

```
statement if Boolean expression;
statement unless Boolean expression;
statement while Boolean expression;
statement until Boolean expression;
statement foreach list;
```

### 3.9 See Also

w:Perl control structures

4 Read files

4.1 Procedural Interface

4.1.1 By slurping file

This method will read the whole file into an array. It will split on the special variable $/

```perl
# Create a read-only file handle for foo.txt
open ( my $fh, '<', 'foo.txt' );

# Read the lines into the array @lines
my @lines=<$fh>;

# Print out the whole array of lines
print @lines;
```

4.1.2 By line processing

This method will read the file one line at a time. This will keep memory usage down, but the program will have to poll the input stream on each iteration.

```perl
# Create a read-only file handle for foo.txt
open ( my $fh, '<', 'foo.txt' );

# Iterate over each line, saving the line to the scalar variable $line
while ( my $line = <$fh> ) {
    # Print out the current line from foo.txt
    print $line;
}
```

4.2 Object Oriented Interface

Using IO::File you can get a more modern object-oriented interface to a perl file handle.

```perl
# Include IO::File which will give you the interface
use IO::File;

# Create a read-only file handle for foo.txt
my $fh = IO::File->new( 'foo.txt', 'r' );

# Iterate over each line, saving the line to the scalar variable $line
while ( my $line = $fh->getline ) {
```

45
# Print out the current line from foo.txt
print $line;
}

# Include IO::File which will give you the interface
use IO::File;

# Create a read-only file handle for foo.txt
my $fh = IO::File->new( 'foo.txt', 'r' );
my @lines = $fh->getlines;

# Print out the current line from foo.txt
print @lines;

In addition to the basic control structures, Perl allows the use of statement modifiers. The statement modifier is placed at the end of the statement that it modifies. Note that the do {} until and do {} while loop constructs are actually statement modifiers. The complete list of modifiers is:

- `statement if expression`
- `statement unless expression`
- `statement while expression`
- `statement until expression`
- `statement foreach list`

Unlike BASIC-PLUS, statement modifiers in Perl cannot be stacked.

### 4.3 String functions

#### 4.3.1 chomp

**Action**

Removes the last characters from a string only if they're recognized as a record separator (e.g. a newline character)

**Returns**

?

**Syntax**

chomp($String = $__);
4.3.2 chop

**Action**

Removes the last character from a string regardless

**Returns**

?

**Syntax**

chop($String = $_);

**Example**

chop; # removes the last character from 
chop(); # (same)
chop($String); # removes the last character from $String

**See Also**

- **chomp**
  - To remove the last character from a string if it is a record separator

Removes the last character from a string (e.g. removes the newline characters when reading from a file)

4.3.3 chr

print chr(65); # Prints a capital A

---

1 Chapter 4.46.2 on page 95
2 Chapter 4.46.1 on page 95
Read files

Gets an ASCII character, given it's code

### 4.3.4 crypt

```perl
# One-way hash function
my $HashedWord = crypt($Word, $Salt);
```

(See also MD5\(^3\) )

The salt string needs only be 2 characters long, and provides a way of randomising the hash, such that the same word can produce several different hashes, if used with different values of $Salt:

### 4.3.5 hex

```perl
print hex(11); # Prints B
```

Converts a number to hexadecimal
Other way around - converts hex to number: `print hex(11); # prints 17`
you can use
`printf("%X",11); # Prints B`

### 4.3.6 index

Search for one string within another. (see `rindex` to search from end-to-start)

```perl
$Result = index($Haystack, $Needle);
$Result = index($Haystack, $Needle, $StartPosition);
index("Some text", "bleh"); # Returns -1 (not found)
index("Some text", "Some"); # Returns 0 (first character)
index("Some text", "text"); # Returns 5 (sixth character)
```

The special variable $[ always gets added to the return value, but $[ is normally 0, and the manual recommends leaving it at 0.

### 4.3.7 lc

```perl
$Lowercase = lc($String);
```

Converts a string to lower-case

---

3 http://search.cpan.org/~gaas/MD5-2.03/MD5.pm
4.3.8 lcfirst
Converts the first character of a string to lowercase

4.3.9 length

```perl
print "String is ", length($String) , " characters long\n";
```

Returns the length of a string

4.3.10 oct

```perl
print oct(8); # Prints 10
```

Converts a number to octal

4.3.11 ord

```perl
print ord("A"); # prints 65
```

Converts a character to its number.

4.3.12 pack

Takes a list and converts it into a string using a supplied set of rules.

```perl
my $String = pack($Template, @ListOfNumbers);
my $String = pack("CCCC",65,66,67,68); # Result: *ABCD*
```

$Template can be made up of:

- `a` A string with arbitrary binary data, will be null padded.
- `A` An ascii string, will be space padded.
- `Z` A null terminated (asciz) string, will be null padded.
- `b` A bit string (ascending bit order inside each byte, like `vec()`).
- `B` A bit string (descending bit order inside each byte).
- `h` A hex string (low nybble first).
- `H` A hex string (high nybble first).
- `c` A signed char value.
- `C` An unsigned char value. Only does bytes. See U for Unicode.
### Read files

<table>
<thead>
<tr>
<th>Letter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>A signed short value.</td>
</tr>
<tr>
<td>S</td>
<td>An unsigned short value. (Exactly 16 bits unless you use the ! suffix)</td>
</tr>
<tr>
<td>i</td>
<td>A signed integer value.</td>
</tr>
<tr>
<td>I</td>
<td>An unsigned integer value. (At least 32 bits wide, machine-dependent)</td>
</tr>
<tr>
<td>l</td>
<td>A signed long value.</td>
</tr>
<tr>
<td>L</td>
<td>An unsigned long value. (Exactly 32 bits unless you use the ! suffix)</td>
</tr>
<tr>
<td>n</td>
<td>An unsigned short in &quot;network&quot; (big-endian) order.</td>
</tr>
<tr>
<td>N</td>
<td>An unsigned long in &quot;network&quot; (big-endian) order.</td>
</tr>
<tr>
<td>v</td>
<td>An unsigned short in &quot;VAX&quot; (little-endian) order.</td>
</tr>
<tr>
<td>V</td>
<td>An unsigned long in &quot;VAX&quot; (little-endian) order. (Exactly 16 bits and 32 bits respectively)</td>
</tr>
<tr>
<td>q</td>
<td>A signed quad (64-bit) value.</td>
</tr>
<tr>
<td>Q</td>
<td>An unsigned quad value. (Only available if your system supports 64-bit integers and Perl has been compiled to support them)</td>
</tr>
<tr>
<td>f</td>
<td>A single-precision float in the native format.</td>
</tr>
<tr>
<td>d</td>
<td>A double-precision float in the native format.</td>
</tr>
<tr>
<td>p</td>
<td>A pointer to a null-terminated string.</td>
</tr>
<tr>
<td>P</td>
<td>A pointer to a structure (fixed-length string).</td>
</tr>
<tr>
<td>u</td>
<td>A uuencoded string.</td>
</tr>
<tr>
<td>U</td>
<td>A Unicode character number. Encodes to UTF-8 internally.</td>
</tr>
<tr>
<td>w</td>
<td>A BER compressed integer. Its bytes represent an unsigned integer in base 128, most significant digit first, with as few digits as possible. Bit eight (the high bit) is set on each byte except the last.</td>
</tr>
<tr>
<td>x</td>
<td>A null byte.</td>
</tr>
<tr>
<td>X</td>
<td>Back up a byte.</td>
</tr>
<tr>
<td>@</td>
<td>Null fill to absolute position.</td>
</tr>
</tbody>
</table>

Each letter may optionally be followed by a number giving a repeat count.

The integer types s, S, l, and L may be immediately followed by a ! suffix to signify native shorts or longs.

### 4.3.13 reverse

Reverses a string (in scalar context) or a list (in list context):

```perl
my @ReversedList = reverse(@List);
```
4.3.14 rindex

Search for one string within another, starting at the end of the string.

```perl
$Result = rindex($Haystack, $Needle);
$Result = rindex($Haystack, $Needle, $StartPosition);
```

```perl
rindex("Some text", "bleh"); # Returns -1 (not found)
rindex("Some text", "Some"); # Returns 0 (first character)
rindex("abbbbbb", "b"); # Returns 5 (first "b" found, when starting at the end)
```

4.3.15 sprintf

Prints a formatted string:

```perl
my $Text = sprintf("%d / %d is %08.5f", 1, 3, 1/3); # Result: "10 / 3 is 003.33333"
```

```perl
sprintf("Character: %c", 65);
sprintf("String %s", "Hello");
sprintf("Signed integer: %d", 15);
sprintf("Unsigned integer: %u", 15);
sprintf("Unsigned int (in octal): %o", 15);
sprintf("Unsigned int (in hex): %x", 15); # Use %X to get upper-case output
sprintf("Binary number: %b", 15);
sprintf("Scientific notation: %e", 5000); # Use %E to get upper-case output
sprintf("Floating point number: %f", 1/3); # 0.3333333
sprintf("Floating point number: %g", 1/3); # Decides between scientific and float. %G is upper-case
sprintf("Pointer: %p", $Variable);
```

Use `%c` to get a percent-sign.

Use `%n` to request the number of characters written so far, and put it into the next variable in the list. You may want to check that user-supplied formatting rules don’t contain this code.
### 4.3.16 substr

Return part of a string (a *substring*)

**Format:** `substr string start-position length`

*start-position* is zero-based.

A negative number starts from the end of the string.

```php
$FirstLetter = substr($Text, 0, 1);  // First letter
$First3Letters = substr($Text, 0, 3); // First three letters
$Last3Letters = substr($Text, -3);   // Last three letters
```

You can use `substr` on the left side of an assignment statement to change part of a string. This can actually shorten or lengthen the string.

```php
$text = 'cat dog';
substr ($mystring, 3, 1) = ' and '; // $text now contains 'cat and dog'
```

### 4.3.17 uc

```php
$Uppercase = uc($String);
```

Converts a string to upper-case

### 4.3.18 ucfirst

Converts the first character of a string to uppercase

### 4.4 Numeric functions

#### 4.4.1 abs

Returns the absolute(positive) value of a number

```php
$Number = abs(-100);  // Returns 100;
```
4.4.2 atan2

```perl
# Converts cartesian(x,y) coordinates into an angle
$Number = atan2($Y, $X);
```

4.4.3 cos

```perl
# Returns the cosine of an angle (radians)
$Number = cos($Angle);  # Cosine = Adjacent/Hypotenuse
```

4.4.4 exp

```perl
# Raises e to a specified power
$Number = exp(2);  # Returns e^2

e ≈ 2.71828183 more about e
```

4.4.5 hex

```perl
# Interprets a string as hexadecimal, and returns its value
$Number = hex("10");  # Returns 16
$Number = hex("0xFF");  # Returns 255
```

4.4.6 int

Rounds a number towards zero, returning an integer

```perl
$Number = int(-1.6);  # Returns -1
$Number = int(0.9);  # Returns 0
$Number = int(28.54);  # Returns 28
```

4.4.7 log

```perl
# Returns the natural logarithm of a number
$Number = log(2.71828183);  # Returns 1
$Number = exp(log($X));  # Returns $X
$Number = log($X) / log(10);  # Returns log10($X). Alternately, you can use the log10() function in the POSIX module
$Number = log($X) / log(15);  # Returns log to the base 15 of $X
```

4.4.8 oct

```perl
# Interprets a string as octal, and returns its value
$Number = oct("10");  # Returns 8
$Number = oct("21");  # Returns 17
```
4.4.9 rand

```perl
# Gets a random number (may automatically call srand() if that's not been done)
$Number = rand();   # Returns a random number from 0 to 1
$Number = int(rand(800));   # Returns a random integer from 0 to 799
$Number = 1 + int(rand(999));   # Returns a random integer from 1 to 999
```

4.4.10 sin

```perl
# Returns the sine of an angle (radians)
$Number = sin($Angle);   # Sine = Opposite/Hypotenuse
```

4.4.11 sqrt

```perl
# Returns the square-root of a number
$Number = sqrt(4);         # Returns 2
$Number = sqrt($X ** 2 + $Y ** 2);   # Returns the diagonal distance across a $X x $Y rectangle

See the Math::Complex module if you need to take roots of negative numbers;
```

4.4.12 srand

```perl
# Seeds (sets-up) the random-number generator
srand();
```

Version-dependant, and older versions of Perl are not guaranteed to have a good seed value. See the Math::TrulyRandom module for more possibilities. The current version of Perl uses the urandom device if it's available.

4.5 Array functions

4.5.1 pop

```perl
$LastElement = pop(@MyArray);
```

Take the last element from an array

4.5.2 push

```perl
push(@MyArray, "Last element");
push(@MyArray, "several", "more", "elements");
```

Push a list of elements onto the end of an array
4.5.3 shift

```perl
shift(@MyArray); # Delete the first element
$FirstElement = shift(@MyArray); # Delete the first element, load it into $FirstElement instead
```

Take the first element out of an array

4.5.4 splice

```perl
# Removes elements from an array, optionally replacing them with a new array
splice(@Array); # Removes all elements from array
splice(@Array, 10); # Removes from element 10 to the end of the array
splice(@Array, -10); # Removes the last 10 elements of the array
splice(@Array, 0, 10); # Removes the first 10 elements of the array
@NewArray = splice(@Array, 0, 10, @Array2); # Replaces the first 10 elements of the array with Array2
```

4.5.5 unshift

```perl
unshift(@MyArray, "New element");
unshift(@MyArray, "several", "more", "elements");
```

Add a list of elements onto the beginning of an array

4.6 List functions

4.6.1 grep

```perl
# Returns a list of elements for which an expression is true
@TextFiles = grep(/\.txt$/i, @AllFiles);
$NumberOfTextFiles = grep(/\.txt$/i, @AllFiles);
# Can use a block of code instead of an expression
@TextFiles = grep({return(substr($_, -3) eq "txt");}, @AllFiles);
```

4.6.2 join

```perl
# Joins the items of a list into a single string
@$OneItemPerLine = join("\n", @List);
@$EverythingBunchedTogether = join("", @List);
@$Filename = join("/", ($Directory, $Subdirectory, $Filename));
```
4.6.3 map

```perl
# Evaluates a block of code for each item in a list, and returns
# a list of the results
@UppercaseList = map(uc, @List);
@Numbers = map { "Number $_" } 1..100;
```

4.6.4 reverse

```perl
# Reverses the order of a list
@ReversedList = reverse(@List);
# In scalar context, concatenates the list and then reverses the
# string
$ReversedString = reverse('foo','bar','baz'); # gives 'zabraboof'
```

4.6.5 sort

```perl
# Sorts the elements in a list
@AsciiSort = sort(@RandomList);
@AsciiSort = sort @RandomList;
foreach $Item (sort @RandomList) {...}
# Can specify a function to decide the sort order
@CaseInsensitiveSort = sort { uc($a) cmp uc($b) } @RandomList;
@NumericSort = sort { $a <=> $b } @RandomList;
@CustomSort = sort custom_function_name @RandomList;
```

4.6.6 unpack

Unpacks a string into a list - see the templates available for the pack() function for details

4.7 Associative array functions

4.7.1 delete

```perl
# Remove an element from a hash
$h = ( 'a'=>1, 'cow'=>'moo', 'b'=>2 );
delete $h{cow};
# $h now contains ( 'a'=>1, 'b'=>2 )
```

4.7.2 each

```perl
# Return the 'next' key/value pair { in a random order }
while ( ( $key, $value ) = each $hash ) {
    print "$key => $value\n";
}
```
4.7.3 exists

```perl
# Tests whether or not a key exists in a hash (even if the value for
# that key is undef)
if (exists $hash{$key}){
   print "\$hash contains a value for key '\$key'\n";
}
```

4.7.4 keys

```perl
# Returns a list of all keys from the hash, in same 'random' order
# as each
foreach $key (keys %hash){
   print "$key => \$hash{$key}\n";
}
```

4.7.5 values

```perl
# Returns a list of all values from the hash, in same 'random' order
# as keys
foreach $value (values %hash){
   print "\$hash contains a value '\$value'\n";
}
```

4.8 Input and output functions

4.8.1 binmode

4.8.2 close

```perl
# closes a filehandle when it is no longer needed
close(STDERR); # hide debugging info from the user
```

4.8.3 closedir

```perl
# Close a directory open by opendir
closedir(DIRHANDLE);
```

4.8.4 dbmclose

4.8.5 dbmopen

4.8.6 die

Exits the program, printing to "STDERR" the first parameter and the current file and line. Used to trap errors.

```perl
die "Error: $!\n" unless chdir '/';
```
eof FILEHANDLE
eof()
eof

This function returns true if the next read on FILEHANDLE would return end-of-file, or if FILEHANDLE is not open. FILEHANDLE may be an expression whose value gives the real filehandle, or a reference to a filehandle object of some sort. An eof without an argument returns the end-of-file status for the last file read. An eof() with empty parentheses () tests the ARGV filehandle (most commonly seen as the null filehandle in <>). Therefore, inside a while (<> ) loop, an eof() with parentheses will detect the end of only the last of a group of files. Use eof (without the parentheses) to test each file in a while (<>) loop. For example, the following code inserts dashes just before the last line of the last file:

```
while (<> ) {
    if (eof() ) {
       print "-" x 30, "\n";
    }
    print;
}
```

On the other hand, this script resets line numbering on each input file:

```
# reset line numbering on each input file
while (<>) {
    next if /^\s*#/;     # skip comments
    print "$\t$_";
} continue {
    close ARGV if eof;   # Not eof()
}
```

Like "$" in a sed program, eof tends to show up in line number ranges. Here's a script that prints lines from /pattern/ to end of each input file:

```
while (<>) {
    print if /pattern/ .. eof;
}
```

Here, the flip-flop operator (..) evaluates the pattern match for each line. Until the pattern matches, the operator returns false. When it finally matches, the operator starts returning true, causing the lines to be printed. When the eof operator finally returns true (at the end of the file being examined), the flip-flop operator resets, and starts returning false again for the next file in @ARGV.
4.8.8 fileno

4.8.9 flock

4.8.10 format

4.8.11 getc

4.8.12 print

Prints the parameters given.

Discussed in the following sections:

Digression on print in Strings section\(^5\)

\(^5\) Chapter 0.7.3 on page 10
4.8.13 printf
4.8.14 read
4.8.15 readdir
4.8.16 rewinddir
4.8.17 seek
4.8.18 seekdir
4.8.19 select
4.8.20 syscall
4.8.21 sysread
4.8.22 sysseek
4.8.23 syswrite
4.8.24 tell
4.8.25 telldir
4.8.26 truncate
4.8.27 warn
4.8.28 write

4.9 Functions for working with fixed length records

4.9.1 pack

See the entry for pack further up the page

4.9.2 read

```c
# Reads data from a file-handle
read(FILEHANDLE, $StoreDataHere, $NumberBytes);
```

```c
# Returns the number of bytes read
$NumberBytesRead = read(FILEHANDLE, $StoreDataHere, $NumberBytes);
```
# Optional offset is applied when the data is stored (not when reading)
read(FILEHANDLE, $StoreDataHere, $NumberBytes, Offset);

## 4.9.3 syscall

# Runs a system command
syscall( $Command, $Argument1, $Argument2, $Argument3);

# (maximum 14 arguments)
$ReturnValue = syscall($Command);

## 4.9.4 sysread

## 4.9.5 syswrite

## 4.9.6 unpack

# See the pack function for details (unpack does the opposite!)
unpack($Template, $BinaryData);

## 4.9.7 vec

### 4.10 Filesystem functions

#### 4.10.1 -X

if( -r $FullFilename) // File is readable by effective uid/gid.
if( -w $FullFilename) // File is writable by effective uid/gid.
if( -x $FullFilename) // File is executable by effective uid/gid.
if( -o $FullFilename) // File is owned by effective uid.

if( -R $FullFilename) // File is readable by real uid/gid.
if( -W $FullFilename) // File is writable by real uid/gid.
if( -X $FullFilename) // File is executable by real uid/gid.
if( -O $FullFilename) // File is owned by real uid.

if( -e $FullFilename) // File exists.
if( -z $FullFilename) // File has zero size.
if( -s $FullFilename) // File has nonzero size (returns size).

if( -f $FullFilename) // File is a plain file.
if( -d $FullFilename) // File is a directory.
if( -l $FullFilename) // File is a symbolic link.
if( -p $FullFilename) // File is a named pipe (FIFO), or Filehandle is a pipe.
if( -S $FullFilename) // File is a socket.
if( -b $FullFilename) // File is a block special file.
if( -c $FullFilename) // File is a character special file.
if( -t $FullFilename) // Filehandle is opened to a tty.
Read files

if(-u $FullFilename) // File has setuid bit set.
if(-g $FullFilename) // File has setgid bit set.
if(-k $FullFilename) // File has sticky bit set.

if(-T $FullFilename) // File is an ASCII text file.
if(-B $FullFilename) // File is a "binary" file (opposite of -T).

$Age = -M $FullFilename; // Age of file in days when script started.
$Age = -A $FullFilename; // Same for access time.
$Age = -C $FullFilename; // Same for inode change time.

4.10.2 chdir

chdir $Directory;
chdir $Directory || die("Couldn't change directory");

4.10.3 chmod

chmod 0744 $File1;
chmod 0666 $File1, $File2, $File3;
# 0 for octal, at the beginning of a number

<table>
<thead>
<tr>
<th>Owner</th>
<th>Group</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Write</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Read</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>======</td>
<td>--</td>
<td>-----</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.10.4 chown

# Change the owner of a file
chown($NewUserID, $NewGroupID, $Filename);
chown($NewUserID, $NewGroupID, $File1, $File2, $File3);

chown($NewUserID, -1, $Filename); # Leave group unchanged
chown(-1, $NewGroupID, $Filename); # Leave user unchanged

4.10.5 chroot

chroot $NewRootDirectory;

Sets the root directory for the program, such that the "/" location refers to the specified directory.

Program must be running as root for this to succeed.
4.10.6 fcntl

4.10.7 glob

```perl
# Expands filenames, in a shell-like way
my @TextFiles = glob("*.txt");
```

See also File::Glob

4.10.8 ioctl

4.10.9 link

```perl
# Creates a link to a file
link($ExistingFile, $LinkLocation);
link($ExistingFile, $LinkLocation) || die("Couldn't create link");
```

4.10.10 lstat

Identical to stat(), except that if given file is symbolic link, stat link not the target.

4.10.11 mkdir

```perl
mkdir $Filename || die("Couldn't create directory");
mkdir $Filename, 0777; # Make directory with particular file-permissions
```

4.10.12 open

```perl
open(my $FileHandle, $Filename) || die("Couldn't open file");
ope($fp, "<", $Filename); # Read from file
open(my $fp, ">", $Filename); # Write to file
open(my $fp, ">>", $Filename); # Append to file
```
4.10.13 opendir

```perl
opendir(my $DirHandle, $Directory) || die("Couldn't open directory");
while (my $Filename = readdir $DirHandle)
  # Do something with $Filename in $Directory
closedir($DirHandle);
```

```perl
opendir(DIR, $Directory) || die("Couldn't open directory");
foreach readdir(DIR){
  # Do something with $_ in $Directory
}
closedir(DIR);
```

4.10.14 readlink

```perl
# Finds the value of a symbolic link
$LinkTarget = readlink($LinkPosition);
```

4.10.15 rename

```perl
rename $OldFile, $NewFile or die("Couldn't move file");
```

May work differently on non-*nix operating systems, and possibly not at all when moving between different filesystems. See

Figure 1

for more complicated file operations.

4.10.16 rmdir

```perl
rmdir $Filename || die("Couldn't remove directory");
```

4.10.17 stat

```perl
@FileStatistics = stat($Filename);
```

```perl
$DeviceNum = $FileStatistics[0]; # device number of filesystem
$Inode = $FileStatistics[1]; # inode number
$FileMode = $FileStatistics[2]; # (type and permissions)
$NumHardLinks = $FileStatistics[3]; # number of (hard) links to the file
$UserID = $FileStatistics[4]; # numeric user ID
$GroupID = $FileStatistics[5]; # numeric group ID
$DeviceIdent = $FileStatistics[6]; # Device identifier (special files only)
$SizeBytes = $FileStatistics[7];
$AccessTime = $FileStatistics[8]; # seconds since the epoch
$ModifyTime = $FileStatistics[9];
```
4.10.18 symlink

```perl
# Creates a new filename symbolically linked to the old filename
symlink($OldFilename, $NewFilename);
symlink($OldFilename, $NewFilename) || die("Couldn't create symlink");
eval(symlink($OldFilename, $NewFilename));
```

4.10.19 umask

```perl
# Sets or returns the umask for the process.
my $UMask = umask();
umask(0000); # This process can create any type of files
umask(0001); # This process can't create world-readable files
umask(0444); # This process can't create executable files
```

4.10.20 unlink

```perl
# Deletes a file
unlink $Filename;
unlink $Filename || die("Couldn't delete file");
unlink $File1, $File2, $File3;
(unlink($File1, $File2, $File3) == 3) || die("Couldn't delete files");
```

4.10.21 utime

```perl
# Updates the modification times of a list of files
my $AccessTime = time();
my $ModificationTime = time();

utime($AccessTime, $ModificationTime, $Filename);
my $NumFilesChanged = utime($AccessTime, $ModificationTime, $File1, $File2, $File3);
```

4.11 Program functions

4.11.1 caller

Returns information about the current function call stack. In scalar context, returns only the name of the package from where the current subroutine was called. In list context, returns the package, filename, and line number. In list context with a numeric argument passed, returns several pieces of information (see below). The argument represents how many levels in the call stack to go back.

```perl
#!/usr/bin/perl
```
Read files

```perl
foo();
sub foo {
  $package = caller; # returns 'main'
  ($package, $filename, $line) = caller; # returns 'main', the file
  # name, and 3
  # Line below returns all 10 pieces of info. (Descriptions
  # self-explanatory from variable names)
  ($package, $filename, $line, $subroutine, $hasargs, $wantarray,
  $evaltext, $is_require, $hints, $bitmask) = caller(0);
}
```

### 4.11.2 import

There is no actual 'import' function. Rather, it is a convention when writing a module to create a subroutine named 'import' which populates the current namespace with that module's needed variables and/or methods.

The standard 'Exporter' module provides an import method if your class has it as a base class.

### 4.11.3 package

Declares all lines that follow (until EOF or the next package statement) to belong to the given package's namespace.

```perl
#!/usr/bin/perl
$x = 5; # sets $main::x
package Foo;
$x = 5; # sets $Foo::x
sub bar { # defines &Foo::bar
  print "hello world";
}
package Temp;
$x = 5; # sets $Temp::x
```

### 4.11.4 require

includes the specified module's code into the current program. The module can be specified either with an absolute or relative path, or with a bareword. If a bareword is given, a '.pm' extention is added, and :: is replaced with the current operating system's path seperator:

```perl
require Foo::Bar;
# identical to:
require 'Foo/Bar.pm';
```
4.11.5 use

Requires and imports the given module or pragma, at compile time. The line

```
use Foo qw/bar baz/;
```

is identical to:

```
BEGIN { 
    require Foo;
    import Foo qw/bar baz/;
}
```

4.12 Misc functions

4.12.1 defined

```bash
#returns true if argument is not undef
$x = 0;
print "X defined\n" if defined $x; #prints
print "Y defined\n" if defined $y; #does not print
```

4.12.2 dump

4.12.3 eval

```perl
eval('$a=30;$b=40;');
print $a,$b;
```

4.12.4 formline

4.12.5 local

```perl
#assigns temporary value to global variable for duration of lexical
scope
$x = 5;
print "x = $x\n"; # 5
{
    local $x = 10;
    print "x = $x\n"; # 10
}
print "x = $x\n"; # 5
```

4.12.6 my

```perl
#creates new lexical (ie, not global) variable
$x = 5; #refers to $main:$x
{
```
Read files

my $x = 10;
print "x = $x\n"; # the lexical - 10
print "main's x = $main::x\n" # the global - 5
}
print "x = $x\n"; #the global, because no lexical in scope - 5

4.12.7 reset

#resets hash's internal pointer, to affect lists returned by each
while ($k, $v = each %h) {
  print "$k = $v\n";
  last if ($i++ == 2);
}
#if another each done here, $k,$v will pick up where they left off.
reset %h
#now each will restart from the beginning.

4.12.8 scalar

#forces scalar context on an array
@sizes = (scalar @foo, scalar @bar);
#creates a list of the sizes of @foo and @bar, rather than the
#elements in @foo and @bar

4.12.9 undef

#undefines an existing variable
$x = 5;
undef $x;
print "x = $x\n" if defined $x; #does not print

4.12.10 wantarray

#returns 'true', 'false', or undef if function that called it was
called in list, scalar, or void context, respectively.
sub fctn {
  my @vals = (5..10);
  if (wantarray) {
    return @vals;
  } elsif (defined wantarray) {
    return $vals[0];
  } else {
    warn "Warning! fctn() called in void context!\n";
  }
}
4.13 Processes

4.13.1 alarm

4.13.2 exec

4.13.3 fork

```perl
# clones the current process, returning 0 if clone, and the process id of the clone if the parent
my $pid = fork();
if ($pid == 0) {
    print "I am a copy of the original\n"
} elsif ($pid == -1) {  
    print "I can't create a clone for some reason!\n"
} else {
    print "I am the original, my clone has a process id of $pid\n"
}
```
4.13.4 getpgrp
4.13.5 getppid
4.13.6 getpriority
4.13.7 kill
4.13.8 pipe
4.13.9 qx/STRING/
4.13.10 setpgid
4.13.11 setpriority
4.13.12 sleep
4.13.13 system
4.13.14 times
4.13.15 wait
4.13.16 waitpid

4.14 Modules
4.14.1 do
4.14.2 import
4.14.3 no
4.14.4 package
4.14.5 require
4.14.6 use

4.15 Classes and objects

See also Perl Objects\(^6\)

\(^6\) Chapter 4.25.3 on page 82
Classes and objects
Read files

4.15.1 bless
4.15.2 dbmclose
4.15.3 dbmopen
4.15.4 package
4.15.5 ref
4.15.6 tie
4.15.7 tied
4.15.8 untie
4.15.9 use

4.16 Sockets

4.16.1 accept
4.16.2 bind
4.16.3 connect
4.16.4 getpeername
4.16.5 getsockname
4.16.6 getsockopt
4.16.7 listen
4.16.8 recv
4.16.9 send
4.16.10 setsockopt
4.16.11 shutdown
4.16.12 socket
4.16.13 socketpair

4.17 Login information

4.17.1 endgrent
4.17.2 endhostent
4.17.3 endnetent
4.17.4 endpwent
4.17.5 getgrent
4.17.6 getgrgid
4.17.7 getgrnam
4.17.8 getlogin
4.17.9 getpwent
4.17.10 getpwnam
4.17.11 getpwuid
4.17.12 setgrent
4.17.13 setpwent
4.19.2 localtime

Converts a timestamp to local time

```
@TimeParts = localtime();
@TimeParts = localtime($Time);
```

4.19.3 time

Returns number of seconds since an epoch (which is system-dependant, but may be Jan 1 1970)
See also ../Time::Hires/

4.19.4 times

```
@CPUTimes = times();
$UserTimeForProcess = $CPUTimes[0];
$SystemTimeForProcess = $CPUTimes[1];
$UserTimeForChildren = $CPUTimes[2];
$SystemTimeForChildren = $CPUTimes[3];
```
4.20 Functions that reverse each other

Some functions in perl reverse or otherwise cancel the effect of each other, so running a string through both of them will produce the same output as the input, for example

```
print ord(chr(1));
```

will echo 1 to standard output,

ord()\(^8\) will convert a character to its number in the character set, while chr()\(^9\) will convert a number to its corresponding character, therefore

in the same way that \(\sqrt{x^2} = x\) and \(\sqrt{x^2} = x\) in Mathematics\(^{10}\) (assuming \(x\) is non-negative), \(\text{ord(chr(1)}) = 1\) and \(\text{chr(ord(1)) = 1}\) in Perl.

List of functions that reverse each other:

- `lc()`\(^{11}\) and `uc()`\(^{12}\)
- `lcfirst()`\(^{13}\) and `ucfirst()`\(^{14}\)
- `ord()`\(^{15}\) and `chr()`\(^{16}\)
- `join()`\(^{17}\) and `split()`\(^{18}\)
- `push()`\(^{19}\) and `pop()`\(^{20}\)
- `unshift()`\(^{21}\) and `shift()`\(^{22}\)

These are a set of eight exercises that can be used to test your ability to write Perl programs. In some cases, these exercises might include material not covered from the textbook; in those cases, you may have to consult your platform documentation to identify a necessary function or otherwise implement one yourself.

- ../Exercise 1/
- ../Exercise 2/
- ../Exercise 3/
- ../Exercise 4/

---

8 Chapter 4.46.11 on page 97
9 Chapter 4.46.3 on page 96
11 Chapter 4.46.7 on page 97
12 Chapter 4.46.17 on page 101
13 Chapter 4.46.8 on page 97
14 Chapter 4.46.18 on page 101
15 Chapter 4.46.11 on page 97
16 Chapter 4.46.3 on page 96
17 Chapter 4.49.2 on page 104
18 Chapter 4.51.28 on page 108
19 Chapter 4.48.2 on page 103
20 Chapter 4.48.1 on page 103
21 Chapter 4.48.5 on page 104
22 Chapter 4.48.3 on page 103
4.21 Section 2: In-depth Perl ideas

- http://perldoc.perl.org/perlstyle.html

4.22 Introduction

So you’ve been plodding along with your perl scripts, fiddling with arrays and hashes and suddenly you realize that you would like to pass a function to another function depending on the data you encounter, or perhaps you would like to get back a hash when you look up an array index. References are the thing for you, allowing you to build and pass around ever more complex data structures.

4.23 Referencing and Dereferencing Syntax

```perl
my $nightmare = "clowns";
my $ref = \$nightmare;
print "I laugh in the face of " . $\{\$ref\} . "\n"
```

Output should be *I laugh in the face of clowns.*

The curly brackets are optional, but generally recommended.

4.24 External links

- http://perldoc.perl.org/perlref.html
- http://perldoc.perl.org/perlreftut.html

Regular expressions are tools for complex searching of text, considered one of the most powerful aspects of the Perl language. A regular expression can be as simple as just the text you want to find, or it can include wildcards, logic, and even sub-programs.

To use regular expressions in perl, use the =~ operator to bind a variable containing your text to the regular expression:

---

27 http://en.wikibooks.org/wiki/..%2FExercise%205%2F
28 http://en.wikibooks.org/wiki/..%2FExercise%206%2F
29 http://en.wikibooks.org/wiki/..%2FExercise%207%2F
30 http://en.wikibooks.org/wiki/..%2FExercise%208%2F
$Haystack =˜ /needle/;

This returns 1 if "needle" is contained within $HayStack, or 0 otherwise.

$Haystack =˜ /needle/i; # The i means "case-insensitive"

$Haystack =˜ /(needle|pin)/; # Either/Or statements

$Haystack =˜ /needle \d/; # "needle 0" to "needle 9"

Regular expression can also be used to modify strings. You can search and replace complex patterns by using the regex format s///

$msg = "perl is ok";
$msg =˜ s/ok/awesome/; # search for the word "ok" and
replace it with "awesome"
($msg is now "perl is awesome")

4.24.1 Match a string

# Shorthand form uses // to quote the regular expression
$Text =˜ /search words/;

# The m function allows you to use your choice of quote marks
$Text =˜ m|search words|;
$Text =˜ m{search words};
$Text =˜ m<search words>;
$Text =˜ m#search words#;

4.24.2 Split a string into parts

# The split function allows you to split a string wherever a regular
expression is matched
@ArrayOfParts = split( /,,/, $Text); # Splits wherever a comma is
found
@ArrayOfParts = split( /\s+/, $Text); # Splits where whitespace is
found
@ArrayOfParts = split( /,\s*/, $Text); # Comma followed by optional
whitespace
@ArrayOfParts = split( /\n/, $Text); # Newline marks where to split

4.24.3 Search and replace a string

# The s function allows you to search and replace within a string.
s(substitute)
$Text =˜ s/search for/replace with/;
$Text =˜ s|search for|replace with|;
$Text =˜ s{search for}{replace with};
# Putting a g (global) at the end, means it replaces all occurrences
# and not just the first.
$Text =~ s/search for/replace with/g;

# As with everything, putting an i (insensitive) at the end ignores
# the differences between
# uppercase and lowercase.
$Text =~ s/search for/replace with/i;

4.24.4 Extracting values from a string

# This function sets the variables $1, $2, $3 ...
# to the information that it has extracted from a string.
$Text =~ m/before(.*)after/;
# So, if $Text was "beforeHelloafter", $1 is now "Hello"
$Text =~ m/bef(.*)bet(.*)aft/;
# This time, if $Text was "befOnebetTwoaft", $1 is now "One" and $2 is "Two"

# It can also be used to extract certain kind of information.
$Text =~ m|[^=]+=(\d+)|;
# If $Text was "id=889", $1 now equals "id" and $2 equals 889.

4.24.5 Regular Expressions with Perl Examples

<table>
<thead>
<tr>
<th>Metacharacter</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
</table>
| .             | Matches an arbitrary character, but not a newline. | $string1 = "Hello World\n";
if ($string1 =~ m/....../) {
  print "$string1 has length >= 5\n";
}

| ( )           | Groups a series of pattern elements to a single element. When you match a pattern within parentheses, you can use any of $1, $2, ... $9 later to refer to the previously matched pattern. | Program:
$string1 = "Hello World\n";
if ($string1 =~ m/\(H..\).(o..)/) {
  print "We matched '$1' and '$2'
";
}
Output:
We matched 'Hel' and 'o W';

| +             | Matches the preceding pattern element one or more times. | $string1 = "Hello World\n";
if ($string1 =~ m/1+/) {
  print "There are one or more consecutive 1's in $string1\n";
} |
<table>
<thead>
<tr>
<th>Metacharacter</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>Matches zero or one times.</td>
<td><code>$string1 = &quot;Hello World\n&quot;; if ($string1 =˜ m/H.?e/) { print &quot;There is an 'H' and a 'e' seperated by &quot;; print &quot;0-1 characters (Ex: He Hoe)\n&quot;; }</code></td>
</tr>
<tr>
<td>?</td>
<td>Matches the *, +, or [M,N]’d regexp that comes before as few times as possible.</td>
<td><code>$string1 = &quot;Hello World\n&quot;; if ($string1 =˜ m/(l+?o)/) { print &quot;The non-greedy match with one or more 'l' followed by an 'o' is 'lo', not 'llo'.\n&quot;; }</code></td>
</tr>
<tr>
<td>*</td>
<td>Matches zero or more times.</td>
<td><code>$string1 = &quot;Hello World\n&quot;; if ($string =˜ m/el*o/) { print &quot;There is a 'e' followed by zero to many 'l' followed by 'o' (eo, elo, ello, elllo)\n&quot;; }</code></td>
</tr>
<tr>
<td>[M,N]</td>
<td>Denotes the minimum M and the maximum N match count.</td>
<td><code>$string1 = &quot;Hello World\n&quot;; if ($string1 =˜ m/l{1,2}/) { print &quot;There exists a substring with at least 1 and at most 2 l's in $string1\n&quot;; }</code></td>
</tr>
<tr>
<td>[...]</td>
<td>Denotes a set of possible matches.</td>
<td><code>$string1 = &quot;Hello World\n&quot;; if ($string1 =˜ m/[aeiou]+/) { print &quot;$string1 contains a one or more vowels\n&quot;; }</code></td>
</tr>
<tr>
<td>[ˆ...]</td>
<td>Matches any character not in the square brackets.</td>
<td><code>$string = &quot;Sky.&quot;; if ($string = /[^aeiou]/) { print &quot;$string doesn't contain any vowels\n&quot;; }</code></td>
</tr>
<tr>
<td></td>
<td>Matches one of the left or right operand.</td>
<td>`$string1 = &quot;Hello World\n&quot;; if ($string1 =˜ m/(Hello</td>
</tr>
<tr>
<td>Metacharacter</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>\b</td>
<td>Matches a word boundary.</td>
<td>$string1 = &quot;Hello World\n&quot;; if ($string1 =~ m/ello\b/) { print &quot;There is a word that ends with &quot;; print &quot;'ello'\n&quot;; } else { print &quot;There are no words that end with&quot;; print &quot;'ello'\n&quot;; }</td>
</tr>
<tr>
<td>\w</td>
<td>Matches alphanumeric, including &quot;_&quot;.</td>
<td>$string1 = &quot;Hello World\n&quot;; if ($string1 =~ m/\w/) { print &quot;There is at least one alphanumeric char in $string1 (A-Z, a-z, 0-9, _-)\n&quot;; }</td>
</tr>
<tr>
<td>\W</td>
<td>Matches a non-alphanumeric character.</td>
<td>$string1 = &quot;Hello World\n&quot;; if ($string1 =~ m/\W/) { print &quot;The space between Hello and &quot;; print &quot;World is not alphanumeric\n&quot;; }</td>
</tr>
<tr>
<td>\s</td>
<td>Matches a whitespace character (space, tab, newline, formfeed)</td>
<td>$string1 = &quot;Hello World\n&quot;; if ($string1 =~ m/\s.*\s/) { print &quot;There are TWO whitespace characters separated by other characters in $string1\n&quot;; }</td>
</tr>
<tr>
<td>\S</td>
<td>Matches anything BUT a whitespace.</td>
<td>$string1 = &quot;Hello World\n&quot;; if ($string1 =~ m/\S.*\S/) { print &quot;There are TWO non-whitespace characters separated by other characters in $string1\n&quot;; }</td>
</tr>
<tr>
<td>\d</td>
<td>Matches a digit, same as [0-9].</td>
<td>$string1 = &quot;99 bottles of beer on the wall.&quot;; if ($string1 =~ m/\d+/) { print &quot;$1 is the first number in '$string1'\n&quot;; } '''Output:''' 99 is the first number in '99 bottles of beer on the wall.'</td>
</tr>
<tr>
<td>\D</td>
<td>Matches a non-digit.</td>
<td>$string1 = &quot;Hello World\n&quot;; if ($string1 =~ m/\D/) { print &quot;There is at least one character in $string1\n&quot;; print &quot;that is not a digit.\n&quot;; }</td>
</tr>
<tr>
<td><strong>Metacharacter</strong></td>
<td><strong>Description</strong></td>
<td><strong>Example</strong></td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>^</code></td>
<td>Matches the beginning of a line or string.</td>
<td><code>$string1 = &quot;Hello World\n&quot;; if ($string1 =~ m/He/) { print &quot;$string1 starts with the characters 'He'\n&quot;; }</code></td>
</tr>
<tr>
<td><code>$</code></td>
<td>Matches the end of a line or string.</td>
<td><code>$string1 = &quot;Hello World\n&quot;; if ($string1 =~ m/rld$/) { print &quot;$string1 is a line or string that ends with 'rld'\n&quot;; }</code></td>
</tr>
</tbody>
</table>

### 4.25 Overview

Perl modules (Files that end with the pm extension) are files of perl code that can be reused from program to program. There is an online repository of perl modules called CPAN (Comprehensive Perl Archive Network) at [http://cpan.org](http://cpan.org). Many of these modules come standard with Perl, but others must be installed as needed.

There are thousands of perl modules that do everything from creating a temporary file to calling Amazon web services. These modules can make it easy to quickly write your application if you know how to find, install, and use the appropriate Perl modules. If you are thinking of writing your own Perl module, the best thing to do is to first search at [http://Search.cpan.org](http://Search.cpan.org) to make sure you are not about to reinvent the wheel.

There are two major styles of Perl modules:

1. **Object-Oriented**
2. **Functional**

Some perl modules use both approaches.

To use an object-oriented Perl module you would do something like this:

```perl
use Foo;
my $foo = Foo->new();
print $foo->bar; # call Foo's bar method and print the output.
```

A functional perl module might get used like this:

```perl
use Foo qw/bar/; # Import the name of the subroutine you want to use.
print bar();
```

### 4.25.1 How to install a Perl module

Find the perl module you want at [http://cpan.org](http://cpan.org), and download the gzipped file. Untar and unzip the file:
tar -zxvf MyModule.tgz

Then cd into the directory, and follow the instructions in the README or INSTALL file. You can also use a command-line program called cpan, if you have it installed:

sudo cpan -imt Module::I::Want

4.25.2 To Write your own Perl Module

Perl modules differ from perl scripts in two key and simple ways. Instead of starting the module with "#!/path/to/perl", you start the file with the following:

package My::Module::Name;

You need to end the module with a true value, so the common practice is to do this at the end of the file:

1;

The following is a valid perl module:

package My::Module::Name;
1;

Example

We create a new file called ExampleModule.pm, and in it have the following code:

package ExampleModule;
use strict;
use base "Exporter";
our @EXPORT = qw/hello_world/;
sub hello_world
{
    print "hello, world!\n";
}
1;

We can test to see if the syntax is valid by running:

perl -c ExampleModule.pm

It will print out "ExampleModule.pm syntax OK" if all is well. Otherwise, you can debug using the messages that are printed out.
Now we can use it in a script to see if it works:

```perl
#!/usr/bin/perl
use ExampleModule;
hello_world();
exit;
```

Voilá! You have made a perl module.

### 4.25.3 Create a CPAN-style Perl module

CPAN-style modules have test suites and a way to build the module into the perl library.

Download and install: Module::Starter from CPAN. Once this is installed, there will be a program called module-starter in your path. To create a new module, do the following from the command line:

```
module-starter --module=My::Module::Name, My::Other::Module::Name,
--author="My Name" --email="myemail@gmail.com"
```

It will then create a set of directories for you, including some shell module files with starter POD documentation. The perl modules will be inside the lib directory inside the directory that is created. These are the files to edit. You can put your tests for the modules into the "t" directory. To install and build the module, you do the following:

```
> perl Makefile.PL
> make
> make test
> sudo make install
```

When Perl was initially developed there was no support at all for Object Orientated (OO) programming. Since Perl 5 OO has been added using the concept of Perl packages (namespaces), an operator called bless, some magic variables (@ISA, AUTOLOAD, UNIVERSAL), the `->` and some strong conventions for supporting inheritance and encapsulation.

An object is created using the `package` keyword. All subroutines declared in that package become object or class methods.

A class instance is created by calling a constructor method which must be provided by the class, by convention this method is called `new()`.

Let's see this constructor.

```perl
package Object;

sub new {
    return bless {}, shift;
}
```
Client code can use this class something like this.

```perl
my $o = Object->new;
$o->setA(10);
print $o->getA;
```

This code prints 10.

Let's look at the `new` constructor in a little more detail:

The first thing is that when a subroutine is called using the `->` notation a new argument is pre-pended to the argument list. It is a string with either the name of the Package or a reference to the object (Object->new() or $o->setA). Until that makes sense you will find OO in Perl very confusing.

To use private variables in objects and have variables names check, you can use a little different approach to create objects.

```perl
package my_class;
use strict;
use warnings;
{
  # All code is enclosed in block context
  my %bar; # All vars are declared as hashes
  sub new {
    my $class = shift;
    my $this = \do{ my $scalar }; # object is a reference to scalar
    (inside out object)
    bless $this, $class;
    return $this;
  }

  sub set_bar {
    my $this = shift;
    $bar{$this} = shift;
  }

  sub get_bar {
    my $this = shift;
    return $bar{$this};
  }
}
```

Now you have good encapsulation - you cannot access object variables directly via $o->{bar} but only using set/get methods. It's also impossible to make mistakes in object variable names, because they are not a hash-keys but normal perl variables, needed to be declared.
We use them the same way like hash-blessed objects:

```perl
my $o = my_class->new();
$o->set_bar(10);
print $o->get_bar();
```

prints 10

### 4.26 Section 3: Interfacing Perl

There are several GUI widget sets available as additions to perl, though the most common is probably Perl/Tk.

- Perl Tk\(^{31}\) (sometimes pTk or ptk) is a collection of modules and code which attempts to wed the simple Tk widget set to perl 5.
- Tcl::Tk\(^{32}\) same as perlTk, but uses existing Tcl/Tk via Tcl, so allowing Tcl widgets
- Tkx\(^{33}\) a different, lightweight, access to Tk via Tcl.
- Gtk\(^{34}\) uses Gtk+, the Gimp Toolkit.
- Gtk2\(^{35}\) uses Gt+ 2.x.
- Gtk3\(^{36}\) uses Gt+ 3.x.
- Qt\(^{37}\) uses the Qt toolkit.
- Wx\(^{38}\) uses the platform independent wxWidgets toolkit.
- Prima\(^{39}\) uses its own toolkit.

#### 4.27 Links

The comp.lang.perl.tk FAQ\(^{40}\)

A huge collection of freely usable perl modules, ranging from advanced mathematics to database connectivity, networking and more, can be downloaded from a network of sites called CPAN. Most or all of the software on CPAN is also available under either the Artistic License, the GPL, or both. CPAN.pm is also the name of the perl module that downloads and installs other perl modules from one of the CPAN mirror sites; such installations can be done with interactive prompts, or can be fully automated.

Look for modules on CPAN\(^{41}\)

\(^{31}\) [http://search.cpan.org/dist/Tk/](http://search.cpan.org/dist/Tk/)
\(^{32}\) [http://search.cpan.org/dist/Tcl-Tk/](http://search.cpan.org/dist/Tcl-Tk/)
\(^{33}\) [http://search.cpan.org/dist/Tkx/](http://search.cpan.org/dist/Tkx/)
\(^{34}\) [http://search.cpan.org/dist/Gtk/](http://search.cpan.org/dist/Gtk/)
\(^{35}\) [http://search.cpan.org/dist/Gtk2/](http://search.cpan.org/dist/Gtk2/)
\(^{36}\) [http://search.cpan.org/dist/Gtk3/](http://search.cpan.org/dist/Gtk3/)
\(^{37}\) [http://search.cpan.org/dist/qt/](http://search.cpan.org/dist/qt/)
\(^{38}\) [http://search.cpan.org/dist/Wx/](http://search.cpan.org/dist/Wx/)
\(^{39}\) [http://search.cpan.org/~karasik/Prima/Prima.pm](http://search.cpan.org/~karasik/Prima/Prima.pm)
\(^{40}\) [http://w4.lns.cornell.edu/~pvhp/ptk/ptkFAQ.html](http://w4.lns.cornell.edu/~pvhp/ptk/ptkFAQ.html)
\(^{41}\) [http://search.cpan.org/](http://search.cpan.org/)
4.28 Installing modules

4.28.1 With ActivePerl (Windows systems)

From a command-line, type the command

```
ppm
```

This will give you a “Perl Package Manager” prompt, which allows you to download and install modules from the internet. For example, to install the Time::HiRes module, type:

```
search time::hires
```

That will give you a list of modules which match your search query. Once you know the module is available and what its exact name is, you can install the module with:

```
install Time::HiRes
```

4.28.2 With Perl

If you’re using a normal version of Perl, the way to activate the package manager is this:

```
perl -MCPAN -e shell;
```

This will load the CPAN module, and let you search for, download, install, and manage the modules on your computer the same as PPM.

4.28.3 With Perl (cpanm)

The Perl module cpanm (CPAN Minus) is another alternative for installing modules from the CPAN library [http://search.cpan.org/~miyagawa/App-cpanminus-1.1001/lib/App/cpanminus.pm](http://search.cpan.org/~miyagawa/App-cpanminus-1.1001/lib/App/cpanminus.pm).

cpanm can be installed and used like this on a UNIX-like system:

```
curl -L "http://cpanmin.us" >cpanm
chmod +x cpanm
./cpanm LWP::Bundle
```

One must have root privileges in order to install module in the system-wide directories, however alternatives exist such as local::lib, which allows regular users to install and use Perl modules in their home folder [http://search.cpan.org/~getty/local-lib-1.006007/lib/local/lib.pm](http://search.cpan.org/~getty/local-lib-1.006007/lib/local/lib.pm).
4.28.4 With Strawberry Perl (Windows systems)

Strawberry Perl also includes the CPAN module, so you can use the command above to activate the package manager.

The start menu, however, also includes a shortcut (with the name of "CPAN Client") so that you don't have to go to a command line to do so.

A number of modules are already included in Strawberry Perl, beyond what comes with a normal version of Perl, or what comes with ActivePerl, so you may wish to check if the module you want is already installed before you start the CPAN client.

4.29 Using a module in your program

To incorporate a module into your program, use the Use keyword:

```perl
use Time::HiRes;
```

You can supply an optional list of the functions you want to use from this module, if you're worried that some of the function names in the module are too similar to functions you're already using:

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

With that done, you can simply use the supplied functions as normal. Most modules have example programs within their documentation, and the best way to start using a module is to copy and adapt one of the example programs.

4.30 Finding documentation

The documentation for each module is installed in your documentation directory when you get a new module, or you can browse documentation on search.cpan.org and perldoc.perl.org.

4.30.1 Unix systems

On Unix systems, the documentation is usually installed as man pages in section 3p, so that the command below will work:

```bash
man 3p Module::Name
```

---

42 http://search.cpan.org/
43 http://perldoc.perl.org/
Contributing your own modules to CPAN

perl doc Module::<Name> will also work.

If you want documentation that is browseable in a web browser, you can install Perl doc::Server as noted below.

4.30.2 Windows systems running ActivePerl

Module documentation is installed as HTML files in ActivePerl. To find those files, try looking in some of the following directories:

- C:\Perl\html\lib
- C:\Perl\html\site\lib

If you're having real trouble finding the HTML documentation for a module, you may be able to read the *.pm perl file yourself for POD comments, or use the pod2html tool yourself to generate the HTML file.

4.30.3 Windows systems running Strawberry Perl

Strawberry Perl does not install module documentation as either manpages or html files. Instead, you can run the perl doc command to display module documentation.

perl doc Module::<Name>

You can also use Perl doc::Server to display module documentation, as illustrated below.

4.30.4 Perl doc::Server

The Perl doc::Server module (which can be installed via CPAN) will provide a local server that will display html files "on the fly" from Perl's documentation and the documentation for installed modules. Install it, and the command

perl doc-server

will be in your path. Run it, and then browse to http://localhost:7375/ in your web browser to see the documentation.

Note that the perl doc-server command must be running to provide the documentation using this method.

4.31 Contributing your own modules to CPAN

In the event that a module you need isn't available on CPAN, the usual answer is to write the module yourself and add it to CPAN. That way, nobody else needs to waste time creating the same functionality that you're already written.
See How to contribute modules to CPAN

4.32 Section 4: CGI and Apache

Assuming you already have an Apache server (or compatible server that reads a shebang! line - more on this in a moment) and a perl installation running, it is fairly simple to start running a perl program on the internet.

First, you must have some way to access the program. Here we will deal with form data and submission, so we will assume that your form code in HTML has a property saying ACTION="programname.cgi".

4.33 The Initial Setup

CGI scripts begin like any other Perl program, with a "shebang", something like:

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

(see Perl Programming/First Programs for details)

Next load the CGI module:

```perl
use CGI;
```

The CGI module makes our work easy because it has pre-programmed functions in it for internet use. Then we must create a handle to CGI - something that allows us to access the functions. We do this with:

```perl
my $query = CGI->new();
```

This means that the variable $query is loading the CGI standard functions.

Now that our program is setup using the CGI module, it should look something like this:

```perl
#!/usr/bin/perl
use CGI;
my $query = CGI->new();
```

So we have a program, it just doesn't do anything yet, and will actually cause a server error because the server has no output or any idea of what kind of output to display even if it had some.

---

44 http://cpan.org/misc/cpan-faq.html#How_contribute_modules
45 Chapter 0.2.3 on page 3

88
4.34 Retrieving Information

Before we tell the server what to do with our output, we need to retrieve our input. To do this, we use the $query variable we declared earlier. Say we have a text box in the form that is named "Name" and we want to find out what was typed there. To do this, we put the following line of code in our program:

```perl
my $Name = $query->param('Name');
```

Now, this line of code introduces us to the param() function (for "parameter"). The param() function can do quite a few handy tricks for us, all of them nice ways to retrieve our variables. It processes all of the http coding so all we get is a nice clean variable. Another note, you aren't required to use $Name as your variable. It's simply more convenient to only remember one name for the same variable. Still, use what's best for you.

4.35 Output

Now we must create our header information. CGI even makes THIS easy for us. Instead of memorizing a bunch of mime-type declarations (which you may do as well), all we must do is type:

```perl
print $query->header();
```

and it prints out our header information. A note about headers. Inside the parenthesis, we may specify parameters like cookies to send to the user's browser. This becomes very useful later. For now we will just stick to the headers.

The last thing you need to put (though the program will run, displaying a blank page without it) is some output. Let's simply have it display the user's name back to him/her. This would look like.

```perl
print " You said your name was: $Name";
```

4.36 The Finished Code

So we now have a complete program that processes a form, using only 6 lines of code. Isn't Perl great? The final code looks like this:

```perl
#!/usr/bin/perl
use CGI;
my $query = new CGI;
my $Name = $query->param('Name');
my $query = $query->param('Name');
print $query->header();
```
print "You said your name was: ", $query->escapeHTML($Name);

When put into perspective, we can see that the $query variable is a very important connection to the CGI module as it tells perl that the function you are referencing belongs to CGI; Again, you may declare any variable name in the place of $query so long as you are consistent, though you will find many developers use $query or $q. Also note the use of the escapeHTML method to avoid any HTML injection problems.

Final note. Make sure you change /usr/bin/perl to the path of your perl installation (assuming that is not it) so perl will execute properly.

4.37 Frameworks

There are a number of CGI frameworks to help with common CGI programming tasks:

- CGI::Application
- Catalyst

**mod_perl** is an optional module for Apache. It embeds a Perl interpreter into the Apache server, so that dynamic content produced by Perl scripts can be served in response to incoming requests, without the significant overhead of re-launching the Perl interpreter for each request. As Lincoln D. Stein defined mod_perl in his words:

> mod_perl is more than CGI scripting on steroids. It is a whole new way to create dynamic content by utilizing the full power of the Apache web server to create stateful sessions, customized user authentication systems, smart proxies and much more. Yet, magically, your old CGI scripts will continue to work and work very fast indeed. With mod_perl you give up nothing and gain so much!

mod_perl can emulate a Common Gateway Interface (CGI) environment, so that existing Perl CGI scripts can benefit from the performance boost without having to be re-written.

Unlike CGI (and most other web application environments), mod_perl provides complete access to the Apache API, allowing programmers to write handlers for all phases in the Apache request cycle, manipulate Apache's internal tables and state mechanisms, share data between Apache processes or threads, alter or extend the Apache configuration file parser, and add Perl code to the configuration file itself, among other things.

---

47  http://cgi-app.org/index.cgi
48  http://www.catalystframework.org/
49  http://en.wikibooks.org/wiki/Perl
4.38 External links

- Main website
- Why mod_perl?
- The magic of mod_perl
- Writing Apache Modules with Perl and C
- The mod_perl Developer's Cookbook
- Practical mod_perl
- mod_perl2 User's Guide

4.39 The Initial Setup

4.40 Retrieving Information

4.41 Output

4.42 The Finished Code

4.43 Section 5: Perl and beyond

Perl 6 will separate parsing and compilation and runtime, making the virtual machine more attractive to developers looking to port other languages to the architecture.

Parrot is the Perl6 runtime, and can be programmed at a low level in Parrot assembly language. Parrot exists in a limited form as of June, 2003, and a small number of languages (Jako, Cola, Basic, Forth and a subset of Perl 6) exist simply to be 'compiled' down to Parrot assembly language opcodes.

While Perl6 is being developed, the best way to stay informed about what's happening is to keep an eye on the front-page of http://www.perl.com/ and look out for articles. As each new language-feature is being developed, it gets discussed on Perl.com and the associated mailing lists, so subscribe to some of those to see glimpses of what Perl6 will be like.

51 http://perl.apache.org/
52 http://www.perl.com/pub/a/2002/02/26/whatismodperl.html
53 http://www.revsys.com/writings/modperl.html
54 http://www.modperl.com/
55 http://www.modperlcookbook.org/
56 http://modperlbook.org/
57 http://modperl2book.org/
58 http://en.wikipedia.org/wiki/Perl%206
59 http://en.wikipedia.org/wiki/Parrot%20virtual%20machine
4.43.1 Obfuscated code

Some people claim Perl stands for 'Pathologically Eclectic Rubbish Lister' due to the high use of meaningful punctuation characters in the language syntax.

In common with C programming language\(^{60}\), obfuscated code competitions are an interesting feature of the Perl culture. Similar to obfuscated code but with a different purpose, Perl Poetry is the practice of writing poems that can actually be compiled by perl. This practice is fairly unique to Perl, due to the large number of regular English words used in the language. New poems can regularly be seen in the Perl Poetry section of perlmonks.org\(^{61}\).

4.43.2 Just another Perl Hacker

Your mission, should you choose to accept it, is to write a one-liner perl script which displays the phrase "Just another Perl hacker," (including the comma, and capitalization as shown). If successful, you win the right to use it as an email signature identifying yourself as a Perl hacker. Entries will be judged on how smart-ass the code is. Around 100 of the first JAPHs and some funky obfu Perl can be seen on CPAN\(^{62}\).

4.43.3 Acme

There's always a place in Perl for odd modules, and one such place is the Acme:: namespace. If you have a module which knows how long a piece of string is, or one which converts your perl script into an image of Che Guevara, post it here.

4.43.4 Golf

Perl is a very compact language. So compact, that some have even create a game around perl's terseness called perlgolf. In perlgolf, you are given a problem to solve. You must solve it in the fewest number of characters possible. A scorecard is kept, and after 18 "holes", a winner is announced.

4.44 Section 6: Sample code

This script counts the number of occurrences of each letter in a file:

```perl
#!/usr/bin/perl
# always enable compiler warnings, as they may highlight potential trouble
use warnings;
# let's ask the compiler to be more strict, make sure we declare our variables etc.
use strict;
```

\(^{60}\) http://en.wikibooks.org/wiki/Programming\%3AC
\(^{61}\) http://www.perlmonks.org/index.pl?node=Perl\%20Poetry
\(^{62}\) http://www.cpan.org/misc/japh
# This statement prompts the user for a filename to read from.
print "What file would you like to read from?\n";

# This statement assigns whatever is given on the standard
# input (usually your keyboard) to a scalar
# variable named $filename and removes the newline character that is
# included by default.
chomp (my $filename = <STDIN>);

# This line opens the file referred to in $filename for input via a
# lexical filehandle
# stored in a scalar variable named "$file".
open my $file, "<", $filename or die "Can't open $filename for reading: $^E\n";

# This loop goes through each line of the file, splits the line into
# separate characters and
# increments the number of occurrences of each letter using a hash
# called "$chars".
my %chars;
while (<$file>) {
  $_ = lc($_); # convert everything to lowercase
  my @characters = split (//, $_); # Store list of characters in
  an array
  foreach @characters { # Ignore all characters except
    letters and numbers
    $chars{$_}++;
  }
}
close $file;

# This loop goes through each letter in the %chars hash and prints a
# report informing the user of
# how many times each letter occurred.
foreach my $key (sort keys %chars) {
  if ($chars{$key} == 1) {
    print "$key appeared once.\n";
  } else {
    print "$key appeared $chars{$key} times.\n";
  }
}

If you executed this program on a file containing the sentence "The quick, brown fox jumps over the
lazy dog.", you would see this as output:

a appeared once.
b appeared once.
c appeared once.
d appeared once.
e appeared 3 times.
f appeared once.
g appeared once.
h appeared 2 times.
i appeared once.
j appeared once.
k appeared once.
l appeared once.
m appeared once.
n appeared once.
o appeared 4 times.
p appeared once.
q appeared once.
Hi-Lo: A simple game written in perl that asks you for a guess between 1 and 100 and tells you if you are too high or low.

```perl
use warnings;
use strict;

$| = 1;

print "Enter number of games to play: ";
chomp(my $Num_Games = <STDIN>);

my $Num_Guesses = 0;
for my $gameno (1 .. $Num_Games) {
    my $number = 1 + int rand 100;
    my $guess;
    do {
        print "Enter guess from 1 to 100: ";
        chomp($guess = <STDIN>);
        ++$Num_Guesses;
        if ($guess < $number) {
            print "Higher!\n";
        } elsif ($guess > $number) {
            print "Lower!\n";
        }
    } until $guess == $number;

    print "Correct!\nAverage guesses per game: ",
    $Num_Guesses / $gameno, "\n\n";
}

print "Games played: $Num_Games\n";
```

63 http://en.wikibooks.org/wiki/Category%3APerl%20Programming
64 http://en.wikibooks.org/wiki/Category%3APerl%20Programming
4.45 Section 7: Reference

4.46 String functions

4.46.1 chomp

Action

Removes the last characters from a string only if they're recognized as a record separator (e.g. a newline character)

Returns

?

Syntax

chomp($String = $_);

Example

chomp; # removes the last character from $_ if it is a record separator
chomp(); # (same)
chomp($String); # removes the last character from $String if it is a record separator

See Also

• chop\textsuperscript{65} - To remove the last character from a string

4.46.2 chop

Action

Removes the last character from a string regardless

Returns

?

\textsuperscript{65} Chapter 4.46.2 on page 95
Read files

Syntax

chop($String = $_);

Example

chop; # removes the last character from $_
chop(); # (same)
chop($String); # removes the last character from $String

See Also

• chomp\(^{66}\) - To remove the last character from a string if it is a record separator

Removes the last character from a string (e.g. removes the newline characters when reading from a file)

4.46.3 \texttt{chr}

\begin{verbatim}
print chr(65); # Prints a capital A
\end{verbatim}

Gets an ASCII character, given it's code

4.46.4 \texttt{crypt}

\begin{verbatim}
# One-way hash function
my $HashedWord = crypt($Word, $Salt);
\end{verbatim}

(See also \texttt{MD5}\(^{67}\))

The salt string needs only be 2 characters long, and provides a way of randomising the hash, such that the same word can produce several different hashes, if used with different values of $Salt;

4.46.5 \texttt{hex}

\begin{verbatim}
print hex(11); # Prints B
\end{verbatim}

Converts a number to hexadecimal

Other way around - converts hex to number: print hex(11); # prints 17

you can use

\begin{verbatim}
print sprintf("%X",11); # Prints B
\end{verbatim}

---

\(^{66}\) Chapter 4.46.1 on page 95
\(^{67}\) http://search.cpan.org/~gaas/MD5-2.03/MD5.pm

96
4.46.6 index

Search for one string within another. (see rindex to search from end-to-start)

```perl
$Result = index($Haystack, $Needle);
$Result = index($Haystack, $Needle, $StartPosition);

index("Some text", "bleh"); # Returns -1 (not found)
index("Some text", "Some"); # Returns 0 (first character)
index("Some text", "text"); # Returns 5 (sixth character)
```

The special variable $[] always gets added to the return value, but $[] is normally 0, and the manual recommends leaving it at 0.

4.46.7 lc

```perl
$Lowercase = lc($String);
```

Converts a string to lower-case

4.46.8 lcfirst

Converts the first character of a string to lowercase

4.46.9 length

```perl
print "String is " . length($String) . " characters long\n";
```

Returns the length of a string

4.46.10 oct

```perl
print oct(8); # Prints 10
```

Converts a number to octal

4.46.11 ord

Converts a character to its number.

```perl
print ord("A"); # prints 65
```
4.46.12 pack

Takes a list and converts it into a string using a supplied set of rules.

my $String = pack($Template, @ListOfNumbers);
my $String = pack("CCCC", 65, 66, 67, 68); # Result: "ABCD"

$template can be made up of:

- a A string with arbitrary binary data, will be null padded.
- A An ascii string, will be space padded.
- Z A null terminated (asciz) string, will be null padded.

- b A bit string (ascending bit order inside each byte, like vec()).
- B A bit string (descending bit order inside each byte).
- h A hex string (low nybble first).
- H A hex string (high nybble first).

- c A signed char value.
- C An unsigned char value. Only does bytes. See U for Unicode.

- s A signed short value.
- S An unsigned short value. (Exactly 16 bits unless you use the ! suffix)

- i A signed integer value.
- I An unsigned integer value. (At least 32 bits wide, machine-dependant)

- l A signed long value.
- L An unsigned long value. (Exactly 32 bits unless you use the ! suffix)

- n An unsigned short in "network" (big-endian) order.
- N An unsigned long in "network" (big-endian) order.
- v An unsigned short in "VAX" (little-endian) order.
- V An unsigned long in "VAX" (little-endian) order. (Exactly 16 bits and 32 bits respectively)

- q A signed quad (64-bit) value.
- Q An unsigned quad value. (Only available if your system supports 64-bit integers and Perl has been compiled to support them)

- f A single-precision float in the native format.
- d A double-precision float in the native format.

- p A pointer to a null-terminated string.
- P A pointer to a structure (fixed-length string).
**u** A uuencoded string.

**U** A Unicode character number. Encodes to UTF-8 internally.

**w** A BER compressed integer. Its bytes represent an unsigned integer in base 128, most significant digit first, with as few digits as possible. Bit eight (the high bit) is set on each byte except the last.

**x** A null byte.

**X** Back up a byte.

**@** Null fill to absolute position.

Each letter may optionally be followed by a number giving a repeat count.

The integer types s, S, l, and L may be immediately followed by a ! suffix to signify native shorts or longs.

### 4.46.13 reverse

Reverses a string (in scalar context) or a list (in list context):

```perl
code
my @ReversedList = reverse(@List);

# As commonly seen in Perl programs:
foreach( reverse( sort( @List )))
{
  ...
}

my $ReversedString = reverse($String);

my @List = ("One ", "two ", "three...");
my $ReversedListAsString = reverse(@List); # Prints "...eerht owt enO"
```

### 4.46.14 rindex

Search for one string within another, starting at the end of the string.

```perl
code
$Result = rindex($Haystack, $Needle);
$Result = rindex($Haystack, $Needle, $StartPosition);

rindex("Some text", "bleh"); # Returns -1 (not found)
rindex("Some text", "Some"); # Returns 0 (first character)
rindex("abbbbbb", "b"); # Returns 5 (first "b" found, when starting at the end)
```
4.46.15 sprintf

Prints a formatted string:

```perl
my $Text = sprintf("%d / %d is %08.5f", 1, 3, 1/3); # Result: "10 / 3 is 003.33333"
```

```perl
sprintf("Character: %c", 65);
sprintf("String %s", "Hello");
sprintf("Signed integer: %d", 15);
sprintf("Unsigned integer: %u", 15);
sprintf("Unsigned int (in octal): %o", 15);
sprintf("Unsigned int (in hex): %x", 15); # Use %X to get upper-case output
sprintf("Binary number: %b", 15);
sprintf("Scientific notation: %e", 5000); # Use %E to get upper-case output
sprintf("Floating point number: %f", 1/3); # 0.3333333
sprintf("Floating point number: %g", 1/3); # Decides between scientific and float. %G is uppercase
sprintf("Pointer: %p", $Variable);
```

Use %% to get a percent-sign.

Use %n to request the number of characters written so far, and put it into the next variable in the list. You may want to check that user-supplied formatting rules don’t contain this code.

```perl
sprintf("%02d", $Minutes); # Forces leading zeros to make the string 2 characters long
sprintf("%1.5f", $Number); # Limits the number of decimal places
```

4.46.16 substr

Return part of a string (a substring)

Format: substr string start-position length

*start-position* is zero-based.

A negative number starts from the end of the string.

```perl
$FirstLetter = substr($Text, 0, 1); # First letter
$First3Letters = substr($Text, 0, 3); # First three letters
$Last3Letters = substr($Text, -3); # Last three letters
```

You can use *substr* on the left side of an assignment statement to change part of a string. This can actually shorten or lengthen the string.

```perl
$text = 'cat dog';
substr ($mystring, 3, 1) = ' and '; # $text now contains 'cat and dog'
```
4.46.17  **uc**

```php
$Uppercase = uc($String);
```

Converts a string to upper-case

4.46.18  **ucfirst**

Converts the first character of a string to uppercase

**4.47  Numeric functions**

4.47.1  **abs**

Returns the absolute(positive) value of a number

```php
$Number = abs(-100); # Returns 100;
```

4.47.2  **atan2**

# Converts cartesian(x,y) coordinates into an angle
```php
$Number = atan2($Y, $X);
```

4.47.3  **cos**

# Returns the cosine of an angle (radians)
```php
$Number = cos($Angle); # Cosine = Adjacent/Hypotenuse
```

4.47.4  **exp**

# Raises e to a specified power
```php
$Number = exp(2); # Returns e^2
```

\[ e \approx 2.71828183 \text{ more about } e^{x} \]

4.47.5  **hex**

# Interprets a string as hexadecimal, and returns its value
```php
$Number = hex("10"); # Returns 16
$Number = hex("0xFF"); # Returns 255
```
4.47.6 int

Rounds a number towards zero, returning an integer

```
$Number = int(-1.6);  # Returns -1
$Number = int(0.9);   # Returns 0
$Number = int(28.54); # Returns 28
```

4.47.7 log

# Returns the natural logarithm of a number
$Number = log(2.71828183);  # Returns 1
$Number = exp(log($X));      # Returns $X
$Number = log($X) / log(10); # Returns log10($X). Alternately, you can use the log10() function in the POSIX module
$Number = log($X) / log(15); # Returns log to the base 15 of $X

4.47.8 oct

# Interprets a string as octal, and returns its value
$Number = oct("10");  # Returns 8
$Number = oct("21");  # Returns 17

4.47.9 rand

# Gets a random number (may automatically call srand() if that's not been done)
$Number = rand();    # Returns a random number from 0 to 1
$Number = int(rand(800)); # Returns a random integer from 0 to 799
$Number = 1 + int(rand(999)); # Returns a random integer from 1 to 999

4.47.10 sin

# Returns the sine of an angle (radians)
$Number = sin($Angle);  # Sine = Opposite/Hypotenuse

4.47.11 sqrt

# Returns the square-root of a number
$Number = sqrt(4);      # Returns 2
$Number = sqrt($X ** 2 + $Y ** 2); # Returns the diagonal distance across a $X x $Y rectangle

See the Math::Complex module if you need to take roots of negative numbers;
### 4.47.12 srand

```perl
# Seeds (sets-up) the random-number generator
srand();
```

Version-dependant, and older versions of Perl are not guaranteed to have a good seed value. See the Math::TrulyRandom module for more possibilities. The current version of Perl uses the urandom device if it's available.

### 4.48 Array functions

#### 4.48.1 pop

```perl
$LastElement = pop(@MyArray);
```

Take the last element from an array

#### 4.48.2 push

```perl
push(@MyArray, "Last element");
push(@MyArray, "several", "more", "elements");
```

Push a list of elements onto the end of an array

#### 4.48.3 shift

```perl
shift(@MyArray); #Delete the first element
$FirstElement = shift(@MyArray); #Delete the first element, load it into $FirstElement instead
```

Take the first element out of an array

#### 4.48.4 splice

```perl
# Removes elements from an array, optionally replacing them with a new array
splice(@Array); # Removes all elements from array
splice(@Array, 10); # Removes from element 10 to the end of the array
splice(@Array, -10); # Removes the last 10 elements of the array
splice(@Array, 0, 10); # Removes the first 10 elements of the array
@NewArray = splice(@Array, 0, 10); # Removes the first 10 elements of the array and returns those 10 items
splice(@Array, 0, 10, @Array2); # Replaces the first 10 elements of the array with Array2
```
4.48.5 unshift

```perl
unshift(@MyArray, "New element");
unshift(@MyArray, "several", "more", "elements");
```

Add a list of elements onto the beginning of an array

4.49 List functions

4.49.1 grep

```perl
# Returns a list of elements for which an expression is true
@TextFiles = grep(/\.txt$/, @AllFiles);
$NumberOfTextFiles = grep(/\.txt$/, @AllFiles);

# Can use a block of code instead of an expression
@TextFiles = grep({return(substr($_, -3) eq "txt");}, @AllFiles);
```

4.49.2 join

```perl
# Joins the items of a list into a single string
$OneItemPerLine = join( "\n", @List);
$EverythingBunchedTogether = join( "", @List);
$Filename = join( "/", ($Directory, $Subdirectory, $Filename));
```

4.49.3 map

```perl
# Evaluates a block of code for each item in a list, and returns
# a list of the results
@UppercaseList = map(uc, @List);
@Numbers = map {"Number $_"} 1..100;
```

4.49.4 reverse

```perl
# Reverses the order of a list
@ReversedList = reverse(@List);
# In scalar context, concatenates the list and then reverses the
# string
$ReversedString = reverse('foo','bar','baz'); # gives 'zabraboof'
```

4.49.5 sort

```perl
# Sorts the elements in a list
@AsciiSort = sort(@RandomList);
@AsciiSort = sort @RandomList;
foreach $Item (sort @RandomList) {
  ...
}

# Can specify a function to decide the sort order
@CaseInsensitiveSort = sort (uc($a) cmp uc($b)) @RandomList;
```
@NumericSort = sort { $a <=> $b } @RandomList;
@CustomSort = sort custom_function_name @RandomList;

4.49.6 unpack

Unpacks a string into a list - see the templates available for the pack() function for details

4.50  Associative array functions

4.50.1 delete

# Remove an element from a hash
$h = ('a' => 1, 'cow' => 'moo', 'b' => 2);
delete $h{cow};
# %h now contains ('a' => 1, 'b' => 2)

4.50.2 each

# Return the 'next' key/value pair (in a random order)
while ({($key, $value) = each (%hash)}){
    print "$key => $value\n";
}

4.50.3 exists

# Tests whether or not a key exists in a hash (even if the value for
# that key is undef)
if (exists $hash{$key}){
    print "%hash contains a value for key '$key'\n";
}

4.50.4 keys

# Returns a list of all keys from the hash, in same 'random' order
# as each
foreach $key (keys %hash){
    print "$key => $hash{$key}\n";
}

4.50.5 values

# Returns a list of all values from the hash, in same 'random' order
# as keys
foreach $value (values %hash){
    print "%hash contains a value '$value'\n";
}
4.51 Input and output functions

4.51.1 binmode

4.51.2 close

```perl
# closes a filehandle when it is no longer needed
close(STDERR); # hide debugging info from the user
```

4.51.3 closedir

```perl
# Close a directory open by opendir
closedir(DIRHANDLE);
```

4.51.4 dbmclose

4.51.5 dbmopen

4.51.6 die

Exits the program, printing to "STDERR" the first parameter and the current file and line. Used to trap errors.

```perl
die "Error: $!
" unless chdir '/';
```

4.51.7 eof

```perl
eof FILEHANDLE
eof()
eof
```

This function returns true if the next read on FILEHANDLE would return end-of-file, or if FILEHANDLE is not open. FILEHANDLE may be an expression whose value gives the real filehandle, or a reference to a filehandle object of some sort. An eof without an argument returns the end-of-file status for the last file read. An eof() with empty parentheses () tests the ARGV filehandle (most commonly seen as the null filehandle in <>). Therefore, inside a while (<> loop, an eof() with parentheses will detect the end of only the last of a group of files. Use eof (without the parentheses) to test each file in a while (<>) loop. For example, the following code inserts dashes just before the last line of the last file:

```perl
while (<>)
{
    if (eof()) {
        print "-" x 30, "\n";
    }
    print;
}
```
On the other hand, this script resets line numbering on each input file:

```perl
# reset line numbering on each input file
while (<>) {
  next if /^[ ]*#/;  # skip comments
  print "$._$";     
} continue {
  close ARGV if eof;  # Not eof()!
}
```

Like "$" in a sed program, `eof` tends to show up in line number ranges. Here's a script that prints lines from /pattern/ to end of each input file:

```perl
while (<>) {
  print if /pattern/ .. eof;
}
```

Here, the flip-flop operator (..) evaluates the pattern match for each line. Until the pattern matches, the operator returns false. When it finally matches, the operator starts returning true, causing the lines to be printed. When the `eof` operator finally returns true (at the end of the file being examined), the flip-flop operator resets, and starts returning false again for the next file in `@ARGV`

### 4.51.8 fileno

### 4.51.9 flock

### 4.51.10 format

### 4.51.11 getc

### 4.51.12 print

Prints the parameters given.

Discussed in the following sections:

*Digression on print in Strings section*\(^{69}\)

---

\(^{69}\) Chapter 0.7.3 on page 10
4.51.13 printf
4.51.14 read
4.51.15 readdir
4.51.16 rewinddir
4.51.17 seek
4.51.18 seekdir
4.51.19 select
4.51.20 syscall
4.51.21 sysread
4.51.22 sysseek
4.51.23 syswrite
4.51.24 tell
4.51.25 telldir
4.51.26 truncate
4.51.27 warn
4.51.28 write

4.52 Functions for working with fixed length records

4.52.1 pack

See the entry for pack further up the page

4.52.2 read

```c
# Reads data from a file-handle
read(FILEHANDLE, $StoreDataHere, $NumberBytes);
```

```c
# Returns the number of bytes read
$NumberBytesRead = read(FILEHANDLE, $StoreDataHere, $NumberBytes);
```
### 4.52.3 syscall

# Runs a system command
syscall( $Command, $Argument1, $Argument2, $Argument3);

# (maximum 14 arguments)
$ReturnValue = syscall($Command);

### 4.52.4 sysread

### 4.52.5 syswrite

### 4.52.6 unpack

# See the pack function for details (unpack does the opposite!)
unpack( $Template, $BinaryData);

### 4.52.7 vec

### 4.53 Filesystem functions

#### 4.53.1 -X

```plaintext
if( -r $FullFilename) // File is readable by effective uid/gid.
if( -w $FullFilename) // File is writable by effective uid/gid.
if( -x $FullFilename) // File is executable by effective uid/gid.
if( -o $FullFilename) // File is owned by effective uid.
if( -R $FullFilename) // File is readable by real uid/gid.
if( -W $FullFilename) // File is writable by real uid/gid.
if( -X $FullFilename) // File is executable by real uid/gid.
if( -O $FullFilename) // File is owned by real uid.
if( -e $FullFilename) // File exists.
if( -z $FullFilename) // File has zero size.
if( -s $FullFilename) // File has nonzero size (returns size).
if( -f $FullFilename) // File is a plain file.
if( -d $FullFilename) // File is a directory.
if( -l $FullFilename) // File is a symbolic link.
if( -p $FullFilename) // File is a named pipe (FIFO), or Filehandle is a pipe.
if( -S $FullFilename) // File is a socket.
if( -b $FullFilename) // File is a block special file.
if( -c $FullFilename) // File is a character special file.
if( -t $FullFilename) // Filehandle is opened to a tty.
```
Read files

```plaintext
if( -u $FullFilename) // File has setuid bit set.
if( -g $FullFilename) // File has setgid bit set.
if( -k $FullFilename) // File has sticky bit set.

if( -T $FullFilename) // File is an ASCII text file.
if( -B $FullFilename) // File is a "binary" file (opposite of -T).

$Age = -M $FullFilename; // Age of file in days when script started.
$Age = -A $FullFilename; // Same for access time.
$Age = -C $FullFilename; // Same for inode change time.
```

4.53.2 chdir

```plaintext
chdir $Directory;
chdir $Directory || die("Couldn't change directory");
```

4.53.3 chmod

```plaintext
chmod 0744 $File1;
chmod 0666 $File1, $File2, $File3;
# 0 for octal, at the beginning of a number

<table>
<thead>
<tr>
<th></th>
<th>Owner</th>
<th>Group</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Write</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Read</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

4.53.4 chown

```plaintext
# Change the owner of a file
chown($NewUserID, $NewGroupID, $Filename);
chown($NewUserID, $NewGroupID, $File1, $File2, $File3);

chown($NewUserID, -1, $Filename); # Leave group unchanged
chown(-1, $NewGroupID, $Filename); # Leave user unchanged
```

4.53.5 chroot

```plaintext
chroot $NewRootDirectory;
```

Sets the root directory for the program, such that the "/" location refers to the specified directory.

Program must be running as root for this to succeed.
4.53.6 `fcntl`

4.53.7 `glob`

```perl
# Expands filenames, in a shell-like way
my @TextFiles = glob("*.txt");
```

See also `File::Glob`

4.53.8 `ioctl`

4.53.9 `link`

```perl
# Creates a link to a file
link($ExistingFile, $LinkLocation);
link($ExistingFile, $LinkLocation) || die("Couldn't create link");
```

4.53.10 `lstat`

Identical to `stat()`, except that if given file is symbolic link, `stat` link not the target.

4.53.11 `mkdir`

```perl
mkdir $Filename || die("Couldn't create directory");
mkdir $Filename, 0777; # Make directory with particular
                      # file-permissions
```

4.53.12 `open`

```perl
open(my $FileHandle, $Filename) || die("Couldn't open file");
open(my $fp, "<", $Filename); # Read from file
open(my $fp, ">", $Filename); # Write to file
open(my $fp, ">>", $Filename); # Append to file
open(my $fp, "<" $Filename); # Read from file
open(my $fp, ">", $Filename); # Write to file
open(my $fp, ">>", $Filename); # Append to file
open(my $fp, "<", ".\ filename with whitespace \0");
open(my $fp, "<", ".\->filename with reserved characters\0");
open(my $fp, "$Program |"); # Read from the output of another
open(my $fp, "| $Program"); # Write to the input of another
open(my $fp, "<", "-"); # Read from standard input
open(my $fp, ">", "-"); # Write to standard output
```
4.53.13 opendir

```perl
c opendir($DirHandle, $Directory) || die("Couldn't open directory");
while (my $Filename = readdir $DirHandle){
    # Do something with $Filename in $Directory
}
closedir($DirHandle);

c opendir(DIR, $Directory) || die("Couldn't open directory");
foreach readdir(DIR)){
    # Do something with $_ in $Directory
}closedir(DIR);
```

4.53.14 readdir

```perl
# Finds the value of a symbolic link
$LinkTarget = readlink($LinkPosition);
```

4.53.15 rename

```perl
rename $OldFile, $NewFile or die("Couldn't move file");
```

May work differently on non-*nix operating systems, and possibly not at all when moving between different filesystems. See Figure 2 for more complicated file operations.

4.53.16 rmdir

```perl
rmdir $Filename || die("Couldn't remove directory");
```

4.53.17 stat

```perl
@FileStatistics = stat($Filename);

$DeviceNum  = $FileStatistics[0];  # device number of filesystem
$Inode      = $FileStatistics[1];  # inode number
$ FileMode  = $FileStatistics[2];  # (type and permissions)
$NumHardLinks = $FileStatistics[3];  # number of (hard) links to the file
$UserID     = $FileStatistics[4];  # numeric user ID
$GroupID    = $FileStatistics[5];  # numeric group ID
$DeviceIdent = $FileStatistics[6];  # Device identifier (special files only)
$SizeBytes  = $FileStatistics[7];
$AccessTime = $FileStatistics[8];  # seconds since the epoch
$ModifyTime = $FileStatistics[9];
```
4.53.18 symlink

# Creates a new filename symbolically linked to the old filename
symlink($OldFilename, $NewFilename);
symlink($OldFilename, $NewFilename) || die("Couldn't create symlink");
eval(symlink($OldFilename, $NewFilename));

4.53.19 umask

# Sets or returns the umask for the process.
my $UMask = umask();
umask(0000); # This process can create any type of files
umask(0001); # This process can't create world-readable files
umask(0444); # This process can't create executable files

4.53.20 unlink

# Deletes a file
unlink $Filename;
unlink $Filename || die("Couldn't delete file");
unlink $File1, $File2, $File3;
(unlink($File1, $File2, $File3) == 3) || die("Couldn't delete files");

4.53.21 utime

# Updates the modification times of a list of files
my $AccessTime = time();
my $ModificationTime = time();

utime($AccessTime, $ModificationTime, $Filename);
my $NumFilesChanged = utime($AccessTime, $ModificationTime, $File1, $File2, $File3);

4.54 Program functions

4.54.1 caller

Returns information about the current function call stack. In scalar context, returns only the name of the package from where the current subroutine was called. In list context, returns the package, filename, and line number. In list context with a numeric argument passed, returns several pieces of information (see below). The argument represents how many levels in the call stack to go back.

#!/usr/bin/perl
foo();
sub foo {
    $package = caller; #returns 'main'
    ($package, $filename, $line) = caller; #returns 'main', the file name, and 3
    # Line below returns all 10 pieces of info. (Descriptions
    self-explanatory from variable names)
    ($package, $filename, $line, $subroutine, $hasargs, $wantarray,
    $evaltext, $is_require, $hints, $bitmask) = 
        caller(0);
}

4.54.2 import

There is no actual 'import' function. Rather, it is a convention when writing a module to create a subroutine named 'import' which populates the current namespace with that module's needed variables and/or methods.

The standard 'Exporter' module provides an import method if your class has it as a base class.

4.54.3 package

Declares all lines that follow (until EOF or the next package statement) to belong to the given package's namespace.

#!/usr/bin/perl

$x = 5; #sets $main::x
package Foo;
$x = 5; #sets $Foo::x
sub bar { #defines &Foo::bar
    print "hello world";
}
package Temp;
$x = 5; #sets $Temp::x

4.54.4 require

includes the specified module's code into the current program. The module can be specified either with an absolute or relative path, or with a bareword. If a bareword is given, a '.pm' extention is added, and :: is replaced with the current operating system's path seperator:

require Foo::Bar;
#identical to:
require 'Foo/Bar.pm';
4.54.5  use

Requires and imports the given module or pragma, at compile time. The line

```perl
use Foo qw/bar baz/;
```

is identical to:

```perl
BEGIN {
  require Foo;
  import Foo qw/bar baz/;
}
```

4.55  Misc functions

4.55.1  defined

```perl
#returns true if argument is not undef
$x = 0;
print "X defined\n" if defined $x; #prints
print "Y defined\n" if defined $y; #does not print
```

4.55.2  dump

4.55.3  eval

```perl
eval('$a=30;$b=40;');
print $a,$b;
```

4.55.4  formline

4.55.5  local

```perl
#assigns temporary value to global variable for duration of lexical
scope
$x = 5;
print "x = $x\n"; # 5
{
  local $x = 10;
  print "x = $x\n"; # 10
}
print "x = $x\n"; # 5
```

4.55.6  my

```perl
#creates new lexical (ie, not global) variable
$x = 5; #refers to $main:$x
{
```
my $x = 10;
print "x = $x\n";  # the lexical - 10
print "main's x = $main::x\n"  # the global - 5
print "x = $x\n";  # the global, because no lexical in scope - 5

4.55.7 reset

# resets hash's internal pointer, to affect lists returned by each
while ($k, $v = each %h){
    print "$k = $v\n";
    last if ($i++ == 2);
}
#if another each done here, $k,$v will pick up where they left off.
reset %h
# now each will restart from the beginning.

4.55.8 scalar

# forces scalar context on an array
@sizes = (scalar @foo, scalar @bar);
# creates a list of the sizes of @foo and @bar, rather than the
# elements in @foo and @bar

4.55.9 undef

# undefines an existing variable
$x = 5;
undef $x;
print "x = $x\n" if defined $x;  # does not print

4.55.10 wantarray

# returns 'true', 'false', or undef if function that called it was
called in list, scalar, or void context, respectively.
sub fctn {
    my @vals = (5..10);
    if (wantarray) {
        return @vals;
    } elsif (defined wantarray) {
        return $vals[0];
    } else {
        warn "Warning! fctn() called in void context!\n";
    }
}
4.56 Processes

4.56.1 alarm

4.56.2 exec

4.56.3 fork

```perl
#clones the current process, returning 0 if clone, and the process
#id of the clone if the parent
my $pid = fork();
if ($pid == 0) {
    print "I am a copy of the original\n";
} elsif ($pid == -1) {
    print "I can't create a clone for some reason!\n";
} else {
    print "I am the original, my clone has a process id of $pid\n";
}
```
4.56.4 getpgrp
4.56.5 getppid
4.56.6 getpriority
4.56.7 kill
4.56.8 pipe
4.56.9 qx/STRING/
4.56.10 setpgrp
4.56.11 setpriority
4.56.12 sleep
4.56.13 system
4.56.14 times
4.56.15 wait
4.56.16 waitpid

4.57 Modules
4.57.1 do
4.57.2 import
4.57.3 no
4.57.4 package
4.57.5 require
4.57.6 use

4.58 Classes and objects

See also Perl Objects\(^{70}\)

---

\(^{70}\) Chapter 4.25.3 on page 82
4.58.1 bless
4.58.2 dbmclose
4.58.3 dbmopen
4.58.4 package
4.58.5 ref
4.58.6 tie
4.58.7 tied
4.58.8 untie
4.58.9 use

4.59 Sockets

4.59.1 accept
4.59.2 bind
4.59.3 connect
4.59.4 getpeername
4.59.5 getsockname
4.59.6 getsockopt
4.59.7 listen
4.59.8 recv
4.59.9 send
4.59.10 setsockopt
4.59.11 shutdown
4.59.12 socket
4.59.13 socketpair

4.60 Login information

4.60.1 endgrent
4.60.2 endhostent
4.60.3 endnetent
4.60.4 endpwent
4.60.5 getgrent
4.60.6 getgrgid
4.60.7 getgrnam
4.60.8 getlogin
4.60.9 getpwent
4.60.10 getpwnam
4.60.11 getpwuid
4.60.12 setgrent
4.60.13 setpwent

4.61 Network information

4.61.1 endprotoent
4.61.2 endservent
4.61.3 gethostbyaddr
4.61.4 gethostbyname
4.61.5 gethostent
4.61.6 getnetbyaddr
4.61.7 getnetbyname
4.61.8 getnetent
4.61.9 getprotobyname
4.61.10 getprotobynumber
4.61.11 getprotoent
4.61.12 getservbyname
4.61.13 getservbyport
4.61.14 getservent
4.61.15 sethostent
4.61.16 setnetent
4.61.17 setprotoent
4.61.18 setservent

4.62 Time and date

4.62.1 gmtime
Converts a timestamp to GMT
4.62.2 localtime

Converts a timestamp to local time

```perl
@TimeParts = localtime();
@TimeParts = localtime($Time);
```

4.62.3 time

Returns number of seconds since an epoch (which is system-dependant, but may be Jan 1 1970)

See also ../Time::Hires/71

4.62.4 times

```perl
@CPUTimes = times();
$UserTimeForProcess  = $CPUTimes[0];
$SystemTimeForProcess = $CPUTimes[1];
$UserTimeForChildren  = $CPUTimes[2];
$SystemTimeForChildren = $CPUTimes[3];
```

71  http://en.wikibooks.org/wiki/..%2FTime%3A%3AHires%2F
4.63 Functions that reverse each other

Some functions in perl reverse or otherwise cancel the effect of each other, so running a string through both of them will produce the same output as the input, for example

```perl
print ord(chr(1));
```

will echo 1 to standard output,

\[ \text{ord()} \] will convert a character to its number in the character set, while \[ \text{chr()} \] will convert a number to its corresponding character, therefore

in the same way that \( \sqrt{x^2} = x \) and \( \sqrt{x^2} = x \) in Mathematics (assuming \( x \) is non-negative), \[ \text{ord(chr(1)) = 1 and chr(ord(1)) = 1 in Perl.} \]

List of functions that reverse each other:

- \( \text{lc()} \) and \( \text{uc()} \)
- \( \text{lcfirst()} \) and \( \text{ucfirst()} \)
- \( \text{ord()} \) and \( \text{chr()} \)
- \( \text{join()} \) and \( \text{split()} \)
- \( \text{push()} \) and \( \text{pop()} \)
- \( \text{unshift()} \) and \( \text{shift()} \)

See [http://search.cpan.org/](http://search.cpan.org/)

Also, try subscribing to the use.perl.org mailing list, which sends out daily summaries of new modules as they're added to CPAN.

- /File Tests
- Perl functions
- /Regular Expressions
- /Database

---

72 Chapter 4.46.11 on page 97
73 Chapter 4.46.3 on page 96
75 Chapter 4.46.7 on page 99
76 Chapter 4.46.17 on page 101
77 Chapter 4.46.8 on page 97
78 Chapter 4.46.18 on page 101
79 Chapter 4.46.11 on page 97
80 Chapter 4.46.3 on page 96
81 Chapter 4.49.2 on page 104
82 Chapter 4.51.28 on page 108
83 Chapter 4.48.2 on page 103
84 Chapter 4.48.1 on page 103
85 Chapter 4.48.5 on page 104
86 Chapter 4.48.3 on page 103
87 [http://use.perl.org](http://use.perl.org)
• /Time and Date/92

4.64 Key Sites

• perl.org93 - the home of the Perl programming language
• perldoc.perl.org94 - Perl documentation
• cpan.org 95 - The Comprehensive Perl Archive Network, a huge repository for Perl modules and scripts
• dev.perl.org/perl696 - Perl 6 development site
  The Parrot virtual machine97

4.65 Community

• Perl Mongers98, Perl User Group Index
• The Perl Monastery99, themed Perl based help site

4.66 Other

• perl.com100 - a Perl blog
• The DMOZ directory101, DMOZ Perl index
• Perl Outsourcing Stats102
• Perl Tutorial103
• Wikipedia:Perl104

92 http://en.wikibooks.org/wiki/%2ftime%20and%20date
93 http://www.perl.org
94 http://perldoc.perl.org/
95 http://www.cpan.org/
96 http://dev.perl.org/perl6/
97 http://www.parrotcode.org
98 http://www.pm.org
99 http://www.perlmonks.org
100 http://www.perl.com
101 http://dmoz.org/Computers/Programming/Languages/Perl/
102 http://www.odesk.com/trends/perl
103 http://www.programmingbulls.com/perl-tutorial
104 http://en.wikipedia.org/wiki/Perl
## 5 Contributors

<table>
<thead>
<tr>
<th>Edits</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>354d²</td>
</tr>
<tr>
<td>26</td>
<td>Adrignola²</td>
</tr>
<tr>
<td>2</td>
<td>Ahy¹³</td>
</tr>
<tr>
<td>2</td>
<td>AlanUS⁴</td>
</tr>
<tr>
<td>4</td>
<td>Albmont⁵</td>
</tr>
<tr>
<td>1</td>
<td>Alecclews⁶</td>
</tr>
<tr>
<td>2</td>
<td>Alksentrs⁷</td>
</tr>
<tr>
<td>3</td>
<td>Alsocal⁸</td>
</tr>
<tr>
<td>6</td>
<td>Amire80⁹</td>
</tr>
<tr>
<td>5</td>
<td>Archier¹⁰</td>
</tr>
<tr>
<td>1</td>
<td>Arkuat¹¹</td>
</tr>
<tr>
<td>5</td>
<td>Bevatron¹²</td>
</tr>
<tr>
<td>9</td>
<td>Bkuhn¹³</td>
</tr>
<tr>
<td>1</td>
<td>Briefaddn¹⁴</td>
</tr>
<tr>
<td>2</td>
<td>Bseidel¹⁵</td>
</tr>
<tr>
<td>6</td>
<td>CSJewell¹⁶</td>
</tr>
<tr>
<td>1</td>
<td>Captain_panda¹⁷</td>
</tr>
<tr>
<td>1</td>
<td>CarsracBot¹⁸</td>
</tr>
<tr>
<td>1</td>
<td>Castaway¹⁹</td>
</tr>
<tr>
<td>7</td>
<td>CharlesClarkson²⁰</td>
</tr>
<tr>
<td>2</td>
<td>ChippendaleMupp²¹</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>Conrad.Irwin[^22]</td>
</tr>
<tr>
<td>1</td>
<td>Cspurrier[^23]</td>
</tr>
<tr>
<td>1</td>
<td>Dallas1278[^24]</td>
</tr>
<tr>
<td>7</td>
<td>Dan Polansky[^25]</td>
</tr>
<tr>
<td>78</td>
<td>Darklama[^26]</td>
</tr>
<tr>
<td>1</td>
<td>DavidCary[^27]</td>
</tr>
<tr>
<td>1</td>
<td>Derbeth[^28]</td>
</tr>
<tr>
<td>3</td>
<td>Dinomite[^29]</td>
</tr>
<tr>
<td>39</td>
<td>Dirk Hünniger[^30]</td>
</tr>
<tr>
<td>1</td>
<td>Dissident[^31]</td>
</tr>
<tr>
<td>3</td>
<td>Dysprosia[^32]</td>
</tr>
<tr>
<td>60</td>
<td>EvanCarroll[^33]</td>
</tr>
<tr>
<td>2</td>
<td>Fishpi[^34]</td>
</tr>
<tr>
<td>1</td>
<td>Flavio Poletti[^35]</td>
</tr>
<tr>
<td>2</td>
<td>Geocachernemesis[^36]</td>
</tr>
<tr>
<td>1</td>
<td>Guanabot[^37]</td>
</tr>
<tr>
<td>1</td>
<td>Isaacto[^38]</td>
</tr>
<tr>
<td>2</td>
<td>Jdstreng[^39]</td>
</tr>
<tr>
<td>14</td>
<td>JendaCPAN[^40]</td>
</tr>
<tr>
<td>1</td>
<td>Jfmantis[^41]</td>
</tr>
<tr>
<td>16</td>
<td>Jguk[^42]</td>
</tr>
<tr>
<td>1</td>
<td>JoaquinFerrero[^43]</td>
</tr>
<tr>
<td>52</td>
<td>Joelnackman[^44]</td>
</tr>
<tr>
<td>1</td>
<td>Jomegat[^45]</td>
</tr>
<tr>
<td>4</td>
<td>Jonathan Webley[^46]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#</th>
<th>Username</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>Justforasecond</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Jędrzej Pełka</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Kamanashisroy</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Karchnu</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>Karl Dickman</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Kayau</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Krischik</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Larsnyg</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>Lichray</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Luke X</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>Mairi</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Malli kv2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Merime</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Mhoram</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>Mickraus</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>Mike Segal</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Mike.lifeguard</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>Mikecx08</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Mjbmrbot</td>
<td>5</td>
</tr>
<tr>
<td>28</td>
<td>MrItty</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Mrajcok</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Msiren</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>Nex3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Nikai</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Nkrypt</td>
<td>1</td>
</tr>
</tbody>
</table>

48 http://en.wikibooks.org/w/index.php?title=J%C4%99drzej_Pe%C5%82ka
60 http://en.wikibooks.org/w/index.php?title=Mhoram
64 http://en.wikibooks.org/w/index.php?title=Mikecx08
Contributors

<table>
<thead>
<tr>
<th>Username</th>
<th>Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ojw</td>
<td>61</td>
</tr>
<tr>
<td>Omegatron</td>
<td>73</td>
</tr>
<tr>
<td>Pengrate</td>
<td>6</td>
</tr>
<tr>
<td>Plaicy</td>
<td>4</td>
</tr>
<tr>
<td>Pnorcks</td>
<td>2</td>
</tr>
<tr>
<td>Poccil</td>
<td>4</td>
</tr>
<tr>
<td>Prolix</td>
<td>6</td>
</tr>
<tr>
<td>QuiteUnusual</td>
<td>3</td>
</tr>
<tr>
<td>RandalSchwartz</td>
<td>28</td>
</tr>
<tr>
<td>Recent Runes</td>
<td>2</td>
</tr>
<tr>
<td>Rich Farmbrough</td>
<td>1</td>
</tr>
<tr>
<td>Rovenhot</td>
<td>3</td>
</tr>
<tr>
<td>Schwern</td>
<td>1</td>
</tr>
<tr>
<td>Sgibson</td>
<td>2</td>
</tr>
<tr>
<td>Shadow42</td>
<td>1</td>
</tr>
<tr>
<td>Shiman bb</td>
<td>2</td>
</tr>
<tr>
<td>Shoecream</td>
<td>1</td>
</tr>
<tr>
<td>Sigma</td>
<td>21</td>
</tr>
<tr>
<td>Snarius</td>
<td>90</td>
</tr>
<tr>
<td>Spoon</td>
<td>1</td>
</tr>
<tr>
<td>Stennie</td>
<td>92</td>
</tr>
<tr>
<td>The bellman</td>
<td>93</td>
</tr>
<tr>
<td>ThorstenStaerk</td>
<td>94</td>
</tr>
<tr>
<td>Thumperward</td>
<td>95</td>
</tr>
<tr>
<td>Toreau</td>
<td>96</td>
</tr>
</tbody>
</table>

[73](http://en.wikibooks.org/w/index.php?title=User:Omegatron)
[84](http://en.wikibooks.org/w/index.php?title=User:Sgibson)
<table>
<thead>
<tr>
<th>Rank</th>
<th>Username</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Ævar Arnfjörð Bjarmason</td>
<td><a href="http://en.wikibooks.org/w/index.php?title=User:%C3%B0ar_Arnf%C3%B6r%2B_Bjarmason">http://en.wikibooks.org/w/index.php?title=User:%C3%B0ar_Arnf%C3%B6r%2B_Bjarmason</a></td>
</tr>
</tbody>
</table>
List of Figures

- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. http://creativecommons.org/licenses/by-sa/3.0/
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. http://creativecommons.org/licenses/by-sa/2.5/
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. http://creativecommons.org/licenses/by-sa/2.0/
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. http://creativecommons.org/licenses/by-sa/1.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/deed.en
- cc-by-2.5: Creative Commons Attribution 2.5 License. http://creativecommons.org/licenses/by/2.5/deed.en
- cc-by-3.0: Creative Commons Attribution 3.0 License. http://creativecommons.org/licenses/by/3.0/deed.en
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.
- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- CFR: Copyright free use.

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses\(^{111}\). Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your web browser.

\(^{111}\) Chapter 6 on page 135
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Loremus Ipsum</td>
<td>None</td>
</tr>
<tr>
<td>0</td>
<td>Loremus Ipsum</td>
<td>None</td>
</tr>
</tbody>
</table>
6.1 GNU GENERAL PUBLIC LICENSE

This License, referred to as the "GPL License" below, freely gives you permission to run, distribute, and/or modify any free program thatmatch its terms and conditions.

6 Licenses
We have designed this License in order to use it for manuals for free software, even those that are quite thick. We call such a manual a "Transparent" copy of the Document, and define it as a verbatim copy, distributed under this License, together with a separate front-cover Text. A "Transparency" means that the work is in its electronic form, is not an aggregate with other works, and can be displayed at will without any interface or the discrimination of interfaces. A Transparency must have at least these properties:

- It must be possible to copy its electronic files exactly; and
- It must be possible to wholly or partly integrate it with other Transparent copies of the same work.

A "Compilation" of the Document means a collection of separate documents which together form a larger document. The "compilation" and its individual documents, in other words, the works comprising it, must together constitute a single integrated whole, which is not fundamental to the significance of the他人用法。The "Document", below, refers to any such manual or work. Any "Variant" of the Document is a copy or adaptation of it, produced by or under the authority of the publisher, designer or other entity authorized for it to produce Variant(s).

A "Section" of the Document is a subsection of it, numbered using a decimal point like 1, 1.1, 1.1.1, etc. You can modify or add a section to the Document, and use it for supporting your application, without permission from the copyright holder(s) of the Document. However, you should be careful not to modify or add to the Document in a way that makes it appear to be part of the original Document, as the original Document itself contains copyright notices and disclaimers that are not applicable to modified versions.

A "Front-Cover Text" means either the explicit text, the name of the author, or the title of the Document. A "Back-Cover Text" means either the text around the outside of the cover page, or some appropriate substitute title. "Cover Text" includes both "Front-Cover Text" and "Back-Cover Text." You can add Front-Cover Text or Back-Cover Text to the Document as appropriate, given the version you are releasing and all other relevant consideration.

The "Invariant Sections" are certain Sections whose titles are designated in the notice that says that the Document is released under this License. These Sections contain material that must not be altered or removed.

The "License" is an agreement between you (the licensee) and the copyright holder(s) (the licensor) of the Document.

A "Cover Text" is either: (a) A Front-Cover Text or a Back-Cover Text; or (b) the name of the publisher if the Document is published by the publisher;

A "Transparent" copy of the Document means a machine-readable copy, reproductions of the Document, and such reproductions in any medium, of the Document in its entirety; any selection or extract from such a cop
The “Corresponding Application Code” for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work. 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL. 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the Library, then you may convey a copy of the modified version:

a) under this License, provided that you make a good faith effort to incorporate the object code and/or source code for the Library with the rest of your work, if any;

b) Accompany the object code with a copy of the GNU GPL and this license document.

c) For a Combined Work that displays copyright notices as required by section 5 of the GNU GPL, but which does not otherwise incorporate the Library, give prominent notice with the combined library that the Library is used in it and that the Library and its uses are covered by this License. 4. Combined Works.

You may convey a Combined Work under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its uses are covered by this License. 5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License. 6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it specifies that a version designated as “Special Version” applies to it, you have the option of following the terms and conditions either of that published version of the GNU Lesser General Public License or of the license terms for that Special Version from time to time.

The new license is given a distinguishing version number. If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.