PERL PROGRAMMING
programming language

www.tutorialspoint.com
About the Tutorial

Perl is a programming language developed by Larry Wall, especially designed for text processing. It stands for Practical Extraction and Report Language. It runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

This tutorial provides a complete understanding on Perl.

Audience

This reference has been prepared for beginners to help them understand the basic to advanced concepts related to Perl Scripting languages.

Prerequisites

Before you start practicing with various types of examples given in this reference, we are making an assumption that you have prior exposure to C programming and Unix Shell.

Copyright & Disclaimer

© Copyright 2015 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com
# Table of Contents

About the Tutorial ........................................................................................................... i
Audience .......................................................................................................................... i
Prerequisites .................................................................................................................... i
Copyright & Disclaimer ............................................................................................... i
Table of Contents ......................................................................................................... ii

## PART 1: PERL – BASICS .............................................................................................. 1

1. **Introduction** ........................................................................................................... 2
   - What is Perl? .............................................................................................................. 2
   - Perl Features ........................................................................................................... 2
   - Perl and the Web ..................................................................................................... 3
   - Perl is Interpreted ................................................................................................... 3

2. **Environment** .......................................................................................................... 4
   - Getting Perl Installation ....................................................................................... 5
   - Install Perl ............................................................................................................. 5
   - Running Perl ......................................................................................................... 7

3. **Syntax Overview** .................................................................................................... 9
   - First Perl Program ................................................................................................. 9
   - Perl File Extension ............................................................................................... 10
   - Comments in Perl .................................................................................................. 10
   - Whitespaces in Perl ............................................................................................... 11
   - Single and Double Quotes in Perl ....................................................................... 12
   - "Here" Documents ............................................................................................... 12
   - Escaping Characters ............................................................................................. 13
   - Perl Identifiers ....................................................................................................... 14

4. **Data Types** ............................................................................................................ 16
   - Numeric Literals .................................................................................................... 16
   - String Literals ....................................................................................................... 17

5. **Variables** ............................................................................................................... 20
   - Creating Variables ............................................................................................... 20
   - Scalar Variables .................................................................................................... 21
   - Array Variables ..................................................................................................... 21
   - Hash Variables ...................................................................................................... 22
   - Variable Context .................................................................................................... 23

6. **Scalars** ................................................................................................................... 25
   - Numeric Scalars ................................................................................................... 25
   - String Scalars ....................................................................................................... 26
   - Scalar Operations ................................................................................................. 27
   - Multiline Strings ................................................................................................... 27
   - V-Strings ................................................................................................................ 28
   - Special Literals ....................................................................................................... 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7. Arrays</td>
<td>31</td>
</tr>
<tr>
<td>Array Creation</td>
<td>32</td>
</tr>
<tr>
<td>Accessing Array Elements</td>
<td>32</td>
</tr>
<tr>
<td>Sequential Number Arrays</td>
<td>33</td>
</tr>
<tr>
<td>Array Size</td>
<td>34</td>
</tr>
<tr>
<td>Adding and Removing Elements in Array</td>
<td>34</td>
</tr>
<tr>
<td>Slicing Array Elements</td>
<td>36</td>
</tr>
<tr>
<td>Replacing Array Elements</td>
<td>37</td>
</tr>
<tr>
<td>Transform Strings to Arrays</td>
<td>37</td>
</tr>
<tr>
<td>Transform Arrays to Strings</td>
<td>38</td>
</tr>
<tr>
<td>Sorting Arrays</td>
<td>39</td>
</tr>
<tr>
<td>The $! Special Variable</td>
<td>40</td>
</tr>
<tr>
<td>Merging Arrays</td>
<td>40</td>
</tr>
<tr>
<td>Selecting Elements from Lists</td>
<td>41</td>
</tr>
<tr>
<td>8. Hashes</td>
<td>43</td>
</tr>
<tr>
<td>Creating Hashes</td>
<td>43</td>
</tr>
<tr>
<td>Accessing Hash Elements</td>
<td>44</td>
</tr>
<tr>
<td>Extracting Slices</td>
<td>44</td>
</tr>
<tr>
<td>Extracting Keys and Values</td>
<td>45</td>
</tr>
<tr>
<td>Checking for Existence</td>
<td>46</td>
</tr>
<tr>
<td>Getting Hash Size</td>
<td>47</td>
</tr>
<tr>
<td>Add and Remove Elements in Hashes</td>
<td>47</td>
</tr>
<tr>
<td>9. If...Else</td>
<td>49</td>
</tr>
<tr>
<td>if statement</td>
<td>50</td>
</tr>
<tr>
<td>if...else statement</td>
<td>52</td>
</tr>
<tr>
<td>if...elsif...else statement</td>
<td>54</td>
</tr>
<tr>
<td>unless statement</td>
<td>55</td>
</tr>
<tr>
<td>unless...else statement</td>
<td>57</td>
</tr>
<tr>
<td>unless...elsif...else statement</td>
<td>59</td>
</tr>
<tr>
<td>switch statement</td>
<td>60</td>
</tr>
<tr>
<td>The $! : Operator</td>
<td>63</td>
</tr>
<tr>
<td>10. Loops</td>
<td>65</td>
</tr>
<tr>
<td>while loop</td>
<td>66</td>
</tr>
<tr>
<td>until loop</td>
<td>68</td>
</tr>
<tr>
<td>for loop</td>
<td>70</td>
</tr>
<tr>
<td>foreach loop</td>
<td>72</td>
</tr>
<tr>
<td>do...while loop</td>
<td>74</td>
</tr>
<tr>
<td>nested loops</td>
<td>75</td>
</tr>
<tr>
<td>Loop Control Statements</td>
<td>78</td>
</tr>
<tr>
<td>next statement</td>
<td>78</td>
</tr>
<tr>
<td>last statement</td>
<td>81</td>
</tr>
<tr>
<td>continue statement</td>
<td>84</td>
</tr>
<tr>
<td>redo statement</td>
<td>85</td>
</tr>
<tr>
<td>goto statement</td>
<td>87</td>
</tr>
<tr>
<td>The Infinite Loop</td>
<td>90</td>
</tr>
<tr>
<td>11. Operators</td>
<td>91</td>
</tr>
</tbody>
</table>
What is an Operator.................................................................91
Perl Arithmetic Operators .....................................................91
Perl Equality Operators .......................................................93
Perl Assignment Operators ...................................................98
Perl Bitwise Operators .........................................................100
Perl Logical Operators .........................................................103
Quote-like Operators ..........................................................104
Miscellaneous Operators .....................................................105
Perl Operators Precedence ...................................................107

12. Date and Time........................................................................110
Current Date and Time ..........................................................110
GMT Time ...........................................................................111
Format Date & Time .............................................................111
Epoch time ...........................................................................112
POSIX Function strftime() ..................................................113

13. Subroutines .........................................................................116
Define and Call a Subroutine ..................................................116
Passing Arguments to a Subroutine .......................................117
Passing Lists to Subroutines ..................................................118
Passing Hashes to Subroutines ..............................................118
Returning Value from a Subroutine .......................................119
Private Variables in a Subroutine .........................................120
Temporary Values via local() ..............................................121
State Variables via state() ....................................................122
Subroutine Call Context .......................................................123

14. References .........................................................................125
Create References ................................................................125
Dereferencing ......................................................................126
Circular References ...............................................................127
References to Functions .......................................................128

15. Formats ..............................................................................130
Define a Format ....................................................................130
Using the Format ................................................................131
Define a Report Header ........................................................133
Define a Pagination .............................................................134
Number of Lines on a Page ..................................................135
Define a Report Footer ........................................................135

16. File I/O ..............................................................................137
Opening and Closing Files ....................................................137
Open Function .....................................................................137
Sysopen Function ................................................................139
Close Function .....................................................................140
Reading and Writing Files ....................................................140
The <FILEHANDL> Operator ..............................................140
getc Function .....................................................................141
read Function .....................................................................141
### 17. Directories
- Display all the Files .......................................................... 146
- Create new Directory ......................................................... 146
- Remove a directory ............................................................ 148
- Change a Directory ............................................................ 148

### 18. Error Handling
- The if statement ..................................................................... 150
- The unless Function ............................................................ 150
- The ternary Operator ........................................................... 151
- The warn Function ............................................................... 151
- The die Function .................................................................... 151
- Errors within Modules ......................................................... 151
- The carp Function ............................................................... 153
- The cluck Function .............................................................. 153
- The croak Function ............................................................. 154
- The confess Function .......................................................... 155

### 19. Special Variables
- Special Variable Types .......................................................... 157
- Global Scalar Special Variables ............................................ 158
- Global Array Special Variables ............................................ 162
- Global Hash Special Variables ............................................. 163
- Global Special Filehandles .................................................. 163
- Global Special Constants .................................................... 163
- Regular Expression Special Variables ................................. 164
- Filehandle Special Variables ............................................... 165

### 20. Coding Standard ............................................................ 166

### 21. Regular Expressions
- The Match Operator ............................................................ 169
- Match Operator Modifiers ................................................... 171
- Matching Only Once ............................................................ 171
- Regular Expression Variables .............................................. 172
- The Substitution Operator ................................................... 172
- Substitution Operator Modifiers .......................................... 173
- The Translation Operator .................................................... 174
- Translation Operator Modifiers .......................................... 174
- More Complex Regular Expressions .................................... 175
- Matching Boundaries .......................................................... 178
- Selecting Alternatives .......................................................... 179
- Grouping Matching ............................................................. 179
- The \G Assertion ................................................................. 180
Regular-expression Examples ................................................................. 181

22. Sending Email .................................................................................. 186
   Using sendmail Utility ................................................................. 186
   Using MIME::Lite Module ............................................................. 187
   Using SMTP Server ................................................................. 190

PART 2: PERL – ADVANCED TOPICS ...................................................... 191

23. Socket Programming .......................................................................... 192
   What is a Socket? .............................................................................. 192
   To Create a Server ............................................................................. 192
   To Create a Client ............................................................................. 192
   Server Side Socket Calls ................................................................. 193
   Client Side Socket Calls ................................................................. 195
   Client - Server Example ................................................................. 196

24. OOP in Perl ....................................................................................... 199
   Object Basics ...................................................................................... 199
   Defining a Class ................................................................................. 199
   Creating and Using Objects ............................................................. 200
   Defining Methods ............................................................................. 201
   Inheritance ....................................................................................... 203
   Method Overriding ........................................................................... 205
   Default Autoloading .......................................................................... 207
   Destructors and Garbage Collection ................................................. 208
   Object Oriented Perl Example ........................................................ 208

25. Database Access ................................................................................. 212
   Architecture of a DBI Application ...................................................... 212
   Notation and Conventions ............................................................... 212
   Database Connection ......................................................................... 213
   INSERT Operation ............................................................................ 214
   Using Bind Values ............................................................................ 214
   READ Operation .............................................................................. 215
   Using Bind Values ............................................................................ 216
   UPDATE Operation ............................................................................ 216
   Using Bind Values ............................................................................ 217
   DELETE Operation ............................................................................ 218
   Using do Statement ........................................................................... 218
   COMMIT Operation ........................................................................... 218
   ROLLBACK Operation ...................................................................... 219
   Begin Transaction ............................................................................ 219
   AutoCommit Option ......................................................................... 219
   Automatic Error Handling .............................................................. 219
   Disconnecting Database ................................................................... 220
   Using NULL Values .......................................................................... 220
   Some Other DBI Functions ............................................................. 221
   Methods Common to All Handles ..................................................... 222
   Interpolated Statements are Prohibited ............................................. 223
26. CGI Programming ................................................................. 225
What is CGI? ................................................................. 225
Web Browsing ................................................................. 225
CGI Architecture Diagram ................................................................. 226
Web Server Support and Configuration ................................................................. 226
First CGI Program ................................................................. 226
Understanding HTTP Header ................................................................. 227
CGI Environment Variables ................................................................. 228
Raise a “File Download” Dialog Box? ................................................................. 230
GET and POST Methods ................................................................. 230
Passing Information using GET Method ................................................................. 231
Simple URL Example: Get Method ................................................................. 231
Simple FORM Example: GET Method ................................................................. 232
Passing Information using POST Method ................................................................. 233
Passing Checkbox Data to CGI Program ................................................................. 234
Passing Radio Button Data to CGI Program ................................................................. 236
Passing Text Area Data to CGI Program ................................................................. 238
Passing Drop Down Box Data to CGI Program ................................................................. 239
Using Cookies in CGI .................................................................................... 241
How It Works ..................................................................................... 241
Setting up Cookies .................................................................................... 241
Retrieving Cookies .................................................................................... 242
CGI Modules and Libraries .......................................................................... 243

27. Packages and Modules ................................................................. 244
What are Packages? .................................................................................. 244
BEGIN and END Blocks ............................................................................. 245
What are Perl Modules? ............................................................................. 246
The Require Function ............................................................................. 247
The Use Function ............................................................................. 247
Create the Perl Module Tree ...................................................................... 248
Installing Perl Module.............................................................................. 249

28. Process Management ........................................................................... 250
Backtick Operator .................................................................................... 250
The system() Function ............................................................................. 251
The fork() Function ............................................................................. 252
The kill() Function ............................................................................. 254

29. Embedded Documentation ........................................................................ 255
What is POD? .................................................................................... 256
POD Examples .................................................................................... 257

30. Functions References ........................................................................... 259
abs ............................................................................. 259
accept ............................................................................. 260
alarm ............................................................................. 260
atan2 ............................................................................. 262
bind ............................................................................. 263
binmode ............................................................................. 263
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>bless</td>
<td>264</td>
</tr>
<tr>
<td>caller</td>
<td>265</td>
</tr>
<tr>
<td>chdir</td>
<td>266</td>
</tr>
<tr>
<td>chmod</td>
<td>267</td>
</tr>
<tr>
<td>chomp</td>
<td>268</td>
</tr>
<tr>
<td>chop</td>
<td>269</td>
</tr>
<tr>
<td>chown</td>
<td>270</td>
</tr>
<tr>
<td>chr</td>
<td>271</td>
</tr>
<tr>
<td>ASCII Table Lookup</td>
<td>272</td>
</tr>
<tr>
<td>chroot</td>
<td>286</td>
</tr>
<tr>
<td>close</td>
<td>286</td>
</tr>
<tr>
<td>closedir</td>
<td>287</td>
</tr>
<tr>
<td>connect</td>
<td>288</td>
</tr>
<tr>
<td>continue</td>
<td>289</td>
</tr>
<tr>
<td>cos</td>
<td>290</td>
</tr>
<tr>
<td>crypt</td>
<td>290</td>
</tr>
<tr>
<td>dbmclose</td>
<td>291</td>
</tr>
<tr>
<td>dbmopen</td>
<td>292</td>
</tr>
<tr>
<td>defined</td>
<td>293</td>
</tr>
<tr>
<td>delete</td>
<td>295</td>
</tr>
<tr>
<td>die</td>
<td>295</td>
</tr>
<tr>
<td>do</td>
<td>296</td>
</tr>
<tr>
<td>dump</td>
<td>297</td>
</tr>
<tr>
<td>each</td>
<td>298</td>
</tr>
<tr>
<td>endgrent</td>
<td>298</td>
</tr>
<tr>
<td>endhostent</td>
<td>299</td>
</tr>
<tr>
<td>endnetent</td>
<td>300</td>
</tr>
<tr>
<td>endprotoent</td>
<td>302</td>
</tr>
<tr>
<td>endpwent</td>
<td>304</td>
</tr>
<tr>
<td>endservent</td>
<td>328</td>
</tr>
<tr>
<td>eof</td>
<td>329</td>
</tr>
<tr>
<td>eval</td>
<td>330</td>
</tr>
<tr>
<td>exec</td>
<td>331</td>
</tr>
<tr>
<td>exists</td>
<td>332</td>
</tr>
<tr>
<td>exit</td>
<td>332</td>
</tr>
<tr>
<td>exp</td>
<td>333</td>
</tr>
<tr>
<td>fcntl</td>
<td>334</td>
</tr>
<tr>
<td>fileno</td>
<td>334</td>
</tr>
<tr>
<td>flock</td>
<td>335</td>
</tr>
<tr>
<td>fork</td>
<td>336</td>
</tr>
<tr>
<td>format</td>
<td>337</td>
</tr>
<tr>
<td>formline</td>
<td>341</td>
</tr>
<tr>
<td>getc</td>
<td>341</td>
</tr>
<tr>
<td>getgrent</td>
<td>342</td>
</tr>
<tr>
<td>getgrgid</td>
<td>345</td>
</tr>
<tr>
<td>getgrnam</td>
<td>347</td>
</tr>
<tr>
<td>gethostbyaddr</td>
<td>349</td>
</tr>
<tr>
<td>gethostbyname</td>
<td>349</td>
</tr>
<tr>
<td>gethostent</td>
<td>350</td>
</tr>
<tr>
<td>getlogin</td>
<td>351</td>
</tr>
</tbody>
</table>

viii
getnetbyaddr.......................................................................................................................... 352
getnetbyname ............................................................................................................................ 353
getnetent ................................................................................................................................... 354
getpeername .................................................................................................................................. 355
getpgid ......................................................................................................................................... 356
getppid .......................................................................................................................................... 357
getpriority ..................................................................................................................................... 358
getprotobyname .......................................................................................................................... 358
getprotobynumber ....................................................................................................................... 359
getprotoent .................................................................................................................................. 360
getpwnam ...................................................................................................................................... 362
getpwnam ...................................................................................................................................... 364
getpwuid ........................................................................................................................................ 366
getservbyname ............................................................................................................................. 367
getservbyport ................................................................................................................................. 368
getservbyport ................................................................................................................................. 369
getsockopt ...................................................................................................................................... 371
getsockopt ...................................................................................................................................... 372
glob .................................................................................................................................................. 373
gmtime ............................................................................................................................................. 374
goto .................................................................................................................................................. 376
grep .................................................................................................................................................. 377
hex .................................................................................................................................................... 378
import .............................................................................................................................................. 379
index ............................................................................................................................................... 380
int ..................................................................................................................................................... 381
ioctl .................................................................................................................................................. 382
join ................................................................................................................................................... 382
keys ................................................................................................................................................... 383
kill .................................................................................................................................................... 384
last .................................................................................................................................................... 385
lc ....................................................................................................................................................... 387
lcfirst ............................................................................................................................................... 387
length .............................................................................................................................................. 388
link ................................................................................................................................................... 389
listen ................................................................................................................................................ 390
local .................................................................................................................................................. 391
localtime ......................................................................................................................................... 392
lock ................................................................................................................................................... 394
log .................................................................................................................................................... 394
lstat ................................................................................................................................................... 395
m...................................................................................................................................................... 397
map ................................................................................................................................................... 398
mkdir ................................................................................................................................................ 399
msgctl .............................................................................................................................................. 399
msgget ............................................................................................................................................ 400
msgrecv .......................................................................................................................................... 401
msgsnd ............................................................................................................................................ 401
my ..................................................................................................................................................... 402
next .................................................................................................................................................... 403
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>setpgid</td>
<td>464</td>
</tr>
<tr>
<td>setpriority</td>
<td>465</td>
</tr>
<tr>
<td>setprotoent</td>
<td>466</td>
</tr>
<tr>
<td>setpwent</td>
<td>468</td>
</tr>
<tr>
<td>setservent</td>
<td>471</td>
</tr>
<tr>
<td>setsockopt</td>
<td>473</td>
</tr>
<tr>
<td>shift</td>
<td>474</td>
</tr>
<tr>
<td>shmctl</td>
<td>475</td>
</tr>
<tr>
<td>shmget</td>
<td>477</td>
</tr>
<tr>
<td>shmread</td>
<td>479</td>
</tr>
<tr>
<td>shwrite</td>
<td>481</td>
</tr>
<tr>
<td>shutdown</td>
<td>483</td>
</tr>
<tr>
<td>sin</td>
<td>484</td>
</tr>
<tr>
<td>sleep</td>
<td>485</td>
</tr>
<tr>
<td>socket</td>
<td>486</td>
</tr>
<tr>
<td>socketpair</td>
<td>492</td>
</tr>
<tr>
<td>sort</td>
<td>493</td>
</tr>
<tr>
<td>splice</td>
<td>494</td>
</tr>
<tr>
<td>split</td>
<td>495</td>
</tr>
<tr>
<td>sprintf</td>
<td>496</td>
</tr>
<tr>
<td>sqrt</td>
<td>496</td>
</tr>
<tr>
<td>srand</td>
<td>497</td>
</tr>
<tr>
<td>stat</td>
<td>498</td>
</tr>
<tr>
<td>study</td>
<td>499</td>
</tr>
<tr>
<td>sub</td>
<td>500</td>
</tr>
<tr>
<td>substr</td>
<td>501</td>
</tr>
<tr>
<td>symlink</td>
<td>502</td>
</tr>
<tr>
<td>syscall</td>
<td>503</td>
</tr>
<tr>
<td>sysopen</td>
<td>504</td>
</tr>
<tr>
<td>sysread</td>
<td>506</td>
</tr>
<tr>
<td>sysseek</td>
<td>506</td>
</tr>
<tr>
<td>system</td>
<td>508</td>
</tr>
<tr>
<td>syswrite</td>
<td>508</td>
</tr>
<tr>
<td>tell</td>
<td>509</td>
</tr>
<tr>
<td>telldir</td>
<td>510</td>
</tr>
<tr>
<td>tie</td>
<td>511</td>
</tr>
<tr>
<td>tied</td>
<td>514</td>
</tr>
<tr>
<td>time</td>
<td>514</td>
</tr>
<tr>
<td>times</td>
<td>514</td>
</tr>
<tr>
<td>tr</td>
<td>515</td>
</tr>
<tr>
<td>truncate</td>
<td>516</td>
</tr>
<tr>
<td>uc</td>
<td>517</td>
</tr>
<tr>
<td>ucfirst</td>
<td>517</td>
</tr>
<tr>
<td>umask</td>
<td>519</td>
</tr>
<tr>
<td>undef</td>
<td>520</td>
</tr>
<tr>
<td>Description</td>
<td>520</td>
</tr>
<tr>
<td>unlink</td>
<td>521</td>
</tr>
<tr>
<td>unpack</td>
<td>522</td>
</tr>
<tr>
<td>unshift</td>
<td>525</td>
</tr>
<tr>
<td>untie</td>
<td>525</td>
</tr>
<tr>
<td>Command</td>
<td>Page</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
</tr>
<tr>
<td>use</td>
<td>527</td>
</tr>
<tr>
<td>utime</td>
<td>529</td>
</tr>
<tr>
<td>values</td>
<td>529</td>
</tr>
<tr>
<td>vec</td>
<td>530</td>
</tr>
<tr>
<td>wait</td>
<td>531</td>
</tr>
<tr>
<td>waitpid</td>
<td>532</td>
</tr>
<tr>
<td>wantarray</td>
<td>533</td>
</tr>
<tr>
<td>warn</td>
<td>534</td>
</tr>
<tr>
<td>write</td>
<td>534</td>
</tr>
<tr>
<td>-X</td>
<td>535</td>
</tr>
<tr>
<td>y</td>
<td>537</td>
</tr>
</tbody>
</table>
Part 1: Perl – Basics
Perl is a general-purpose programming language originally developed for text manipulation and now used for a wide range of tasks including system administration, web development, network programming, GUI development, and more.

What is Perl?

- Perl is a stable, cross platform programming language.
- Though Perl is not officially an acronym but few people used it as Practical Extraction and Report Language.
- It is used for mission critical projects in the public and private sectors.
- Perl is an Open Source software, licensed under its Artistic License, or the GNU General Public License (GPL).
- Perl was created by Larry Wall.
- Perl 1.0 was released to usenet's alt.comp.sources in 1987.
- At the time of writing this tutorial, the latest version of perl was 5.16.2.
- Perl is listed in the Oxford English Dictionary.

PC Magazine announced Perl as the finalist for its 1998 Technical Excellence Award in the Development Tool category.

Perl Features

- Perl takes the best features from other languages, such as C, awk, sed, sh, and BASIC, among others.
- Perl's database integration interface DBI supports third-party databases including Oracle, Sybase, Postgres, MySQL and others.
- Perl works with HTML, XML, and other mark-up languages.
- Perl supports Unicode.
- Perl is Y2K compliant.
- Perl supports both procedural and object-oriented programming.
- Perl interfaces with external C/C++ libraries through XS or SWIG.
Perl

- Perl is extensible. There are over 20,000 third party modules available from the Comprehensive Perl Archive Network (CPAN).
- The Perl interpreter can be embedded into other systems.

Perl and the Web

- Perl used to be the most popular web programming language due to its text manipulation capabilities and rapid development cycle.
- Perl is widely known as "the duct-tape of the Internet".
- Perl can handle encrypted Web data, including e-commerce transactions.
- Perl can be embedded into web servers to speed up processing by as much as 2000%.
- Perl's mod_perl allows the Apache web server to embed a Perl interpreter.
- Perl's DBI package makes web-database integration easy.

Perl is Interpreted

Perl is an interpreted language, which means that your code can be run as is, without a compilation stage that creates a non portable executable program.

Traditional compilers convert programs into machine language. When you run a Perl program, it's first compiled into a byte code, which is then converted (as the program runs) into machine instructions. So it is not quite the same as shells, or Tcl, which are strictly interpreted without an intermediate representation.

It is also not like most versions of C or C++, which are compiled directly into a machine dependent format. It is somewhere in between, along with Python and awk and Emacs .elc files.
Before we start writing our Perl programs, let's understand how to setup our Perl environment. Perl is available on a wide variety of platforms:

- Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX etc.)
- Win 9x/NT/2000/
- WinCE
- Macintosh (PPC, 68K)
- Solaris (x86, SPARC)
- OpenVMS
- Alpha (7.2 and later)
- Symbian
- Debian GNU/kFreeBSD
- MirOS BSD
- And many more...

This is more likely that your system will have perl installed on it. Just try giving the following command at the $ prompt:

```
$perl -v
```

If you have perl installed on your machine, then you will get a message something as follows:

```
This is perl 5, version 16, subversion 2 (v5.16.2) built for i686-linux

Copyright 1987-2012, Larry Wall

Perl may be copied only under the terms of either the Artistic License or the
GNU General Public License, which may be found in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists, should be found on
this system using "man perl" or "perldoc perl". If you have access to the
```
Internet, point your browser at http://www.perl.org/, the Perl Home Page.

If you do not have perl already installed, then proceed to the next section.

**Getting Perl Installation**

The most up-to-date and current source code, binaries, documentation, news, etc. are available at the official website of Perl.

**Perl Official Website**: http://www.perl.org/

You can download Perl documentation from the following site.

**Perl Documentation Website**: http://perldoc.perl.org

**Install Perl**

Perl distribution is available for a wide variety of platforms. You need to download only the binary code applicable for your platform and install Perl.

If the binary code for your platform is not available, you need a C compiler to compile the source code manually. Compiling the source code offers more flexibility in terms of choice of features that you require in your installation.

Here is a quick overview of installing Perl on various platforms.

**Unix and Linux Installation**

Here are the simple steps to install Perl on Unix/Linux machine.

- Open a Web browser and go to http://www.perl.org/get.html.
- Follow the link to download zipped source code available for Unix/Linux.
- Download `perl-5.x.y.tar.gz` file and issue the following commands at `$` prompt.

```
$tar -xzf perl-5.x.y.tar.gz
$cd perl-5.x.y
$./Configure -de
$make
$make test
$make install
```

**NOTE:** Here `$` is a Unix prompt where you type your command, so make sure you are not typing `$` while typing the above mentioned commands.
This will install Perl in a standard location /usr/local/bin, and its libraries are installed in /usr/local/lib/perlXX, where XX is the version of Perl that you are using.

It will take a while to compile the source code after issuing the `make` command. Once installation is done, you can issue `perl -v` command at $ prompt to check perl installation. If everything is fine, then it will display message like we have shown above.

**Windows Installation**

Here are the steps to install Perl on Windows machine.

- Follow the link for the Strawberry Perl installation on Windows [http://strawberryperl.com](http://strawberryperl.com).
- Download either 32bit or 64bit version of installation.
- Run the downloaded file by double-clicking it in Windows Explorer. This brings up the Perl install wizard, which is really easy to use. Just accept the default settings, wait until the installation is finished, and you're ready to roll!

**Macintosh Installation**

In order to build your own version of Perl, you will need 'make', which is part of the Apples developer tools usually supplied with Mac OS install DVDs. You do not need the latest version of Xcode (which is now charged for) in order to install make.

Here are the simple steps to install Perl on Mac OS X machine.

- Open a Web browser and go to [http://www.perl.org/get.html](http://www.perl.org/get.html).
- Follow the link to download zipped source code available for Mac OS X.
- Download `perl-5.x.y.tar.gz` file and issue the following commands at $ prompt.

```bash
$tar -xzf perl-5.x.y.tar.gz
$cd perl-5.x.y
$./Configure -de
$make
$make test
$make install
```
This will install Perl in a standard location /usr/local/bin, and its libraries are installed in /usr/local/lib/perlXX, where XX is the version of Perl that you are using.

**Running Perl**

The following are the different ways to start Perl.

**1. Interactive Interpreter**

You can enter `perl` and start coding right away in the interactive interpreter by starting it from the command line. You can do this from Unix, DOS, or any other system, which provides you a command-line interpreter or shell window.

```
$perl  -e <perl code>           # Unix/Linux
```

or

```
C:>perl -e <perl code>          # Windows/DOS
```

Here is the list of all the available command line options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-d[:debugger]</td>
<td>Runs program under debugger</td>
</tr>
<tr>
<td>-Idirectory</td>
<td>Specifies @INC/#include directory</td>
</tr>
<tr>
<td>-T</td>
<td>Enables tainting checks</td>
</tr>
<tr>
<td>-t</td>
<td>Enables tainting warnings</td>
</tr>
<tr>
<td>-U</td>
<td>Allows unsafe operations</td>
</tr>
<tr>
<td>-w</td>
<td>Enables many useful warnings</td>
</tr>
<tr>
<td>-W</td>
<td>Enables all warnings</td>
</tr>
<tr>
<td>-X</td>
<td>Disables all warnings</td>
</tr>
<tr>
<td>-e program</td>
<td>Runs Perl script sent in as program</td>
</tr>
<tr>
<td>file</td>
<td>Runs Perl script from a given file</td>
</tr>
</tbody>
</table>

## 2. Script from the Command-line

A Perl script is a text file, which keeps perl code in it and it can be executed at the command line by invoking the interpreter on your application, as in the following:

```bash
$perl script.pl  # Unix/Linux
```

or

```bash
C:>perl script.pl # Windows/DOS
```

## 3. Integrated Development Environment

You can run Perl from a graphical user interface (GUI) environment as well. All you need is a GUI application on your system that supports Perl. You can download Padre, the Perl IDE. You can also use Eclipse Plugin EPIC - Perl Editor and IDE for Eclipse if you are familiar with Eclipse.

Before proceeding to the next chapter, make sure your environment is properly setup and working perfectly fine. If you are not able to setup the environment properly then you can take help from your system administrator.

All the examples given in subsequent chapters have been executed with v5.16.2 version available on the CentOS flavor of Linux.
Perl borrows syntax and concepts from many languages: awk, sed, C, Bourne Shell, Smalltalk, Lisp and even English. However, there are some definite differences between the languages. This chapter is designed to quickly get you up to speed on the syntax that is expected in Perl.

A Perl program consists of a sequence of declarations and statements, which run from the top to the bottom. Loops, subroutines, and other control structures allow you to jump around within the code. Every simple statement must end with a semicolon (;).

Perl is a free-form language: you can format and indent it however you like. Whitespace serves mostly to separate tokens, unlike languages like Python where it is an important part of the syntax, or Fortran where it is immaterial.

**First Perl Program**

**Interactive Mode Programming**

You can use Perl interpreter with `-e` option at command line, which lets you execute Perl statements from the command line. Let's try something at `$` prompt as follows:

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

This execution will produce the following result:

Hello, world

**Script Mode Programming**

Assuming you are already on `$` prompt, let's open a text file `hello.pl` using vi or vim editor and put the following lines inside your file.

```
#!/usr/bin/perl

# This will print "Hello, World"
print "Hello, world\n";
```

Here `/usr/bin/perl` is actual the perl interpreter binary. Before you execute your script, be sure to change the mode of the script file and give execution
privilege, generally a setting of 0755 works perfectly and finally you execute the above script as follows:

```bash
$chmod 0755 hello.pl
$./hello.pl
```

This execution will produce the following result:

```
Hello, world
```

You can use parentheses for functions arguments or omit them according to your personal taste. They are only required occasionally to clarify the issues of precedence. Following two statements produce the same result.

```perl
print("Hello, world\n");
print "Hello, world\n";
```

## Perl File Extension

A Perl script can be created inside of any normal simple-text editor program. There are several programs available for every type of platform. There are many programs designd for programmers available for download on the web.

As a Perl convention, a Perl file must be saved with a .pl or .PL file extension in order to be recognized as a functioning Perl script. File names can contain numbers, symbols, and letters but must not contain a space. Use an underscore (_) in places of spaces.

## Comments in Perl

Comments in any programming language are friends of developers. Comments can be used to make program user friendly and they are simply skipped by the interpreter without impacting the code functionality. For example, in the above program, a line starting with hash # is a comment.

Simply saying comments in Perl start with a hash symbol and run to the end of the line:

```perl
# This is a comment in perl
```

Lines starting with = are interpreted as the start of a section of embedded documentation (pod), and all subsequent lines until the next =cut are ignored by the compiler. Following is the example:

```perl
#!/usr/bin/perl
```
# This is a single line comment
print "Hello, world\n";

=begin comment
This is all part of multiline comment.
You can use as many lines as you like
These comments will be ignored by the
compiler until the next =cut is encountered.
=cut

This will produce the following result:
Hello, world

Whitespaces in Perl

A Perl program does not care about whitespaces. Following program works perfectly fine:

#!/usr/bin/perl

print "Hello, world\n";

But if spaces are inside the quoted strings, then they would be printed as is. For example:

#!/usr/bin/perl

# This would print with a line break in the middle
print "Hello
    world\n";

This will produce the following result:
Hello
    world

All types of whitespace like spaces, tabs, newlines, etc. are equivalent for the interpreter when they are used outside of the quotes. A line containing only
whitespace, possibly with a comment, is known as a blank line, and Perl totally ignores it.

### Single and Double Quotes in Perl

You can use double quotes or single quotes around literal strings as follows:

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

print "Hello, world\n";
print 'Hello, world\n';
```

This will produce the following result:

```
Hello, world
Hello, world\n$
```

There is an important difference in single and double quotes. Only double quotes **interpolate** variables and special characters such as newlines `\n`, whereas single quote does not interpolate any variable or special character. Check below example where we are using `$a` as a variable to store a value and later printing that value:

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

$a = 10;
print "Value of a = $a\n";
print 'Value of a = $a\n';
```

This will produce the following result:

```
Value of a = 10
Value of a = $a\n$
```

### "Here" Documents

You can store or print multiline text with a great comfort. Even you can make use of variables inside the "here" document. Below is a simple syntax, check carefully there must be no space between the `<<` and the identifier.

An identifier may be either a bare word or some quoted text like we used EOF below. If identifier is quoted, the type of quote you use determines the
treatment of the text inside the here document, just as in regular quoting. An unquoted identifier works like double quotes.

```perl
#!/usr/bin/perl
$a = 10;
$var = <<"EOF"
This is the syntax for here document and it will continue until it encounters a EOF in the first line.
This is case of double quote so variable value will be interpolated. For example value of a = $a
EOF
print "$var\n";

$var = <<'EOF';
This is case of single quote so variable value will not be interpolated. For example value of a = $a
EOF
print "$var\n";
```

This will produce the following result:

```
This is the syntax for here document and it will continue until it encounters a EOF in the first line.
This is case of double quote so variable value will be interpolated. For example value of a = 10

This is case of single quote so variable value will be interpolated. For example value of a = $a
```

**Escaping Characters**

Perl uses the backslash (\) character to escape any type of character that might interfere with our code. Let's take one example where we want to print double quote and $ sign:

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

```perl
$result = "This is \"number\"";
print "$result\n";
print "\$result\n";
```

This will produce the following result:

```
This is "number"
$result
```

### Perl Identifiers

A Perl identifier is a name used to identify a variable, function, class, module, or other object. A Perl variable name starts with either $, @ or % followed by zero or more letters, underscores, and digits (0 to 9).

Perl does not allow punctuation characters such as @, $, and % within identifiers. Perl is a **case sensitive** programming language. Thus `$Manpower` and `$manpower` are two different identifiers in Perl.
Perl is a loosely typed language and there is no need to specify a type for your data while using in your program. The Perl interpreter will choose the type based on the context of the data itself.

Perl has three basic data types: scalars, arrays of scalars, and hashes of scalars, also known as associative arrays. Here is a little detail about these data types.

<table>
<thead>
<tr>
<th>S.N.</th>
<th>Types and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Scalar:</strong></td>
</tr>
<tr>
<td></td>
<td>Scalars are simple variables. They are preceded by a dollar sign ($). A scalar is either a number, a string, or a reference. A reference is actually an address of a variable, which we will see in the upcoming chapters.</td>
</tr>
<tr>
<td>2</td>
<td><strong>Arrays:</strong></td>
</tr>
<tr>
<td></td>
<td>Arrays are ordered lists of scalars that you access with a numeric index, which starts with 0. They are preceded by an &quot;at&quot; sign (@).</td>
</tr>
<tr>
<td>3</td>
<td><strong>Hashes:</strong></td>
</tr>
<tr>
<td></td>
<td>Hashes are unordered sets of key/value pairs that you access using the keys as subscripts. They are preceded by a percent sign (%).</td>
</tr>
</tbody>
</table>

**Numeric Literals**

Perl stores all the numbers internally as either signed integers or double-precision floating-point values. Numeric literals are specified in any of the following floating-point or integer formats:

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>1234</td>
</tr>
<tr>
<td>Negative integer</td>
<td>-100</td>
</tr>
</tbody>
</table>
### String Literals

Strings are sequences of characters. They are usually alphanumeric values delimited by either single (’) or double (”) quotes. They work much like UNIX shell quotes where you can use single quoted strings and double quoted strings.

Double-quoted string literals allow variable interpolation, and single-quoted strings are not. There are certain characters when they are proceeded by a back slash, have special meaning and they are used to represent like newline (\n) or tab (\t).

You can embed newlines or any of the following Escape sequences directly in your double quoted strings:

<table>
<thead>
<tr>
<th>Escape sequence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;</td>
<td>Double quote</td>
</tr>
<tr>
<td>'</td>
<td>Single quote</td>
</tr>
<tr>
<td>\a</td>
<td>Alert or bell</td>
</tr>
<tr>
<td>\b</td>
<td>Backspace</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed</td>
</tr>
<tr>
<td>\n</td>
<td>Newline</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return</td>
</tr>
<tr>
<td>Escape</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal tab</td>
</tr>
<tr>
<td>\v</td>
<td>Vertical tab</td>
</tr>
<tr>
<td>\0nn</td>
<td>Creates Octal formatted numbers</td>
</tr>
<tr>
<td>\xnn</td>
<td>Creates Hexadecimal formatted numbers</td>
</tr>
<tr>
<td>\cX</td>
<td>Controls characters, x may be any character</td>
</tr>
<tr>
<td>\u</td>
<td>Forces next character to uppercase</td>
</tr>
<tr>
<td>\l</td>
<td>Forces next character to lowercase</td>
</tr>
<tr>
<td>\U</td>
<td>Forces all following characters to uppercase</td>
</tr>
<tr>
<td>\L</td>
<td>Forces all following characters to lowercase</td>
</tr>
<tr>
<td>\Q</td>
<td>Backslash all following non-alphanumeric characters</td>
</tr>
<tr>
<td>\E</td>
<td>End \U, \L, or \Q</td>
</tr>
</tbody>
</table>

**Example**

Let's see again how strings behave with single quotation and double quotation. Here we will use string escapes mentioned in the above table and will make use of the scalar variable to assign string values.

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

# This is case of interpolation.
$str = "Welcome to \ntutorialspoint.com!";
print "$str\n";

# This is case of non-interpolation.
$str = 'Welcome to \ntutorialspoint.com!';
print "$str\n";
```
# Only W will become upper case.
$str = "\uwelcome to tutorialspoint.com!";
print "$str\n";

# Whole line will become capital.
$str = "\\UWelcome to tutorialspoint.com!";
print "$str\n";

# A portion of line will become capital.
$str = "Welcome to \Ututorialspoint\E.com!";
print "$str\n";

# Backslash non alpha-numeric including spaces.
$str = "\\QWelcome to tutorialspoint's family";
print "$str\n";

This will produce the following result:

Welcome to
tutorialspoint.com!
Welcome to \ntutorialspoint.com!
Welcome to tutorialspoint.com!
WELCOME TO TUTORIALSPOINT.COM!
Welcome to TUTORIALSPOINT.com!
Welcome to\ tutorialspoint's\ family
Variables are the reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or strings in these variables.

We have learnt that Perl has the following three basic data types:

- Scalars
- Arrays
- Hashes

Accordingly, we are going to use three types of variables in Perl. A **scalar** variable will precede by a dollar sign ($) and it can store either a number, a string, or a reference. An **array** variable will precede by sign @ and it will store ordered lists of scalars. Finally, the **Hash** variable will precede by sign % and will be used to store sets of key/value pairs.

Perl maintains every variable type in a separate namespace. So you can, without fear of conflict, use the same name for a scalar variable, an array, or a hash. This means that $foo and @foo are two different variables.

**Creating Variables**

Perl variables do not have to be explicitly declared to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

Keep a note that this is mandatory to declare a variable before we use it if we use **use strict** statement in our program.

The operand to the left of the = operator is the name of the variable, and the operand to the right of the = operator is the value stored in the variable. For example:

```perl
$age = 25;       # An integer assignment
$name = "John Paul"; # A string
$salary = 1445.50; # A floating point
```
Here 25, "John Paul" and 1445.50 are the values assigned to $age, $name and $salary variables, respectively. Shortly we will see how we can assign values to arrays and hashes.

**Scalar Variables**

A scalar is a single unit of data. That data might be an integer number, floating point, a character, a string, a paragraph, or an entire web page. Simply saying it could be anything, but only a single thing.

Here is a simple example of using scalar variables:

```perl
#!/usr/bin/perl
$age = 25;           # An integer assignment
$name = "John Paul"; # A string
$salary = 1445.50;   # A floating point

print "Age = \$age\n";
print "Name = \$name\n";
print "Salary = \$salary\n";
```

This will produce the following result:

```
Age = 25
Name = John Paul
Salary = 1445.5
```

**Array Variables**

An array is a variable that stores an ordered list of scalar values. Array variables are preceded by an "at" (@) sign. To refer to a single element of an array, you will use the dollar sign ($) with the variable name followed by the index of the element in square brackets.

Here is a simple example of using array variables:

```perl
#!/usr/bin/perl
@ages = (25, 30, 40);
@names = ("John Paul", "Lisa", "Kumar");
```
Perl

Here we used escape sign (\) before the $ sign just to print it. Other Perl will understand it as a variable and will print its value. When executed, this will produce the following result:

```perl
$ages[0] = 25
$ages[1] = 30
$ages[2] = 40
$names[0] = John Paul
$names[1] = Lisa
$names[2] = Kumar
```

**Hash Variables**

A hash is a set of **key/value** pairs. Hash variables are preceded by a percent (%) sign. To refer to a single element of a hash, you will use the hash variable name followed by the "key" associated with the value in curly brackets.

Here is a simple example of using hash variables:

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

%data = ('John Paul', 45, 'Lisa', 30, 'Kumar', 40);

print "\$data{'John Paul'} = \$data{'John Paul'}\n";
print "\$data{'Lisa'} = \$data{'Lisa'}\n";
print "\$data{'Kumar'} = \$data{'Kumar'}\n";
```

This will produce the following result:

```perl
$data{'John Paul'} = 45
$data{'Lisa'} = 30
```
Variable Context
Perl treats same variable differently based on Context, i.e., situation where a variable is being used. Let's check the following example:

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

@names = ('John Paul', 'Lisa', 'Kumar');

@copy = @names;
$size = @names;

print "Given names are : @copy\n";
print "Number of names are : $size\n";
```

This will produce the following result:

```
Given names are : John Paul Lisa Kumar
Number of names are : 3
```

Here @names is an array, which has been used in two different contexts. First we copied it into another array, i.e., list, so it returned all the elements assuming that context is list context. Next we used the same array and tried to store this array in a scalar, so in this case it returned just the number of elements in this array assuming that context is scalar context. Following table lists down the various contexts:

<table>
<thead>
<tr>
<th>S.N.</th>
<th>Context and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Scalar:</strong> Assignment to a scalar variable evaluates the right-hand side in a scalar context.</td>
</tr>
<tr>
<td>2</td>
<td><strong>List:</strong> Assignment to an array or a hash evaluates the right-hand side in a list context.</td>
</tr>
<tr>
<td></td>
<td><strong>Boolean:</strong></td>
</tr>
<tr>
<td>---</td>
<td>---------------</td>
</tr>
<tr>
<td></td>
<td>Boolean context is simply any place where an expression is being evaluated to see whether it's true or false.</td>
</tr>
<tr>
<td></td>
<td><strong>Void:</strong></td>
</tr>
<tr>
<td></td>
<td>This context not only doesn't care what the return value is, it doesn't even want a return value.</td>
</tr>
<tr>
<td></td>
<td><strong>Interpolative:</strong></td>
</tr>
<tr>
<td></td>
<td>This context only happens inside quotes, or things that work like quotes.</td>
</tr>
</tbody>
</table>
A scalar is a single unit of data. That data might be an integer number, floating point, a character, a string, a paragraph, or an entire web page.

Here is a simple example of using scalar variables:

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

$age = 25;       # An integer assignment
$name = "John Paul";  # A string
$salary = 1445.50; # A floating point

print "Age = $age\n";
print "Name = $name\n";
print "Salary = $salary\n";
```

This will produce the following result:

```
Age = 25
Name = John Paul
Salary = 1445.5
```

### Numeric Scalars

A scalar is most often either a number or a string. Following example demonstrates the usage of various types of numeric scalars:

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

$integer = 200;
$negative = -300;
$floating = 200.340;
$bigfloat = -1.2E-23;

# 377 octal, same as 255 decimal
```
Perl

```perl
$octal = 0377;

# FF hex, also 255 decimal
$hexa = 0xff;

print "integer = $integer\n";
print "negative = $negative\n";
print "floating = $floating\n";
print "bigfloat = $bigfloat\n";
print "octal = $octal\n";
print "hexa = $hexa\n";
```

This will produce the following result:

```
integer = 200
negative = -300
floating = 200.34
bigfloat = -1.2e-23
octal = 255
hexa = 255
```

---

**String Scalars**

Following example demonstrates the usage of various types of string scalars. Notice the difference between single quoted strings and double quoted strings:

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

$var = "This is string scalar!";
$quote = 'I m inside single quote - $var';
$double = "This is inside single quote - $var";

$escape = "This example of escape -\tHello, World!";

print "var = $var\n";
print "quote = $quote\n";
```

print "double = $double\n";
print "escape = $escape\n";

This will produce the following result:

var = This is string scalar!
quote = I m inside single quote - $var
double = This is inside single quote - This is string scalar!
escape = This example of escape - Hello, World!

**Scalar Operations**

You will see a detail of various operators available in Perl in a separate chapter, but here we are going to list down few numeric and string operations.

```perl
#!/usr/bin/perl
$str = "hello" . "world"; # Concatenates strings.
$num = 5 + 10; # adds two numbers.
$mul = 4 * 5; # multiplies two numbers.
$mix = $str . $num; # concatenates string and number.

print "str = $str\n";
print "num = $num\n";
print "mix = $mix\n";
```

This will produce the following result:

str = helloworld
num = 15
mix = helloworld15

**Multiline Strings**

If you want to introduce multiline strings into your programs, you can use the standard single quotes as below:

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

This will produce the following result:
```perl
$string = 'This is a multiline string';

print "$string\n";
```

This will produce the following result:

```
This is a multiline string
```

You can use "here" document syntax as well to store or print multilines as below:

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

print <<EOF;
This is a multiline string
EOF
```

This will also produce the same result:

```
This is a multiline string
```

## V-Strings

A literal of the form v1.20.300.4000 is parsed as a string composed of characters with the specified ordinals. This form is known as v-strings.

A v-string provides an alternative and more readable way to construct strings, rather than use the somewhat less readable interpolation form "\x{1}\x{14}\x{12c}\x{fa0}".

They are any literal that begins with a v and is followed by one or more dot-separated elements. For example:
#!/usr/bin/perl

$smile = v9786;
$foo = v102.111.111;
$martin = v77.97.114.116.105.110;

print "smile = $smile\n";
print "foo = $foo\n";
print "martin = $martin\n";

This will also produce the same result:

smile = a?q
foo = foo
martin = Martin

Wide character in print at /tmp/135911788320439.pl line 7.

Special Literals
So far you must have a feeling about string scalars and its concatenation and interpolation opration. So let me tell you about three special literals __FILE__, __LINE__, and __PACKAGE__ represent the current filename, line number, and package name at that point in your program.

They may be used only as separate tokens and will not be interpolated into strings. Check the below example:

#!/usr/bin/perl

print "File name __FILE__ .\n";
print "Line Number __LINE__ .\n";
print "Package __PACKAGE__ .\n";

# they cannot be interpolated
print __FILE__ __LINE__ __PACKAGE__ .\n";

This will produce the following result:

File name hello.pl
Line Number 4
Package main
__FILE__ __LINE__ __PACKAGE__
An array is a variable that stores an ordered list of scalar values. Array variables are preceded by an "at" (@) sign. To refer to a single element of an array, you will use the dollar sign ($) with the variable name followed by the index of the element in square brackets.

Here is a simple example of using the array variables:

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

@ages = (25, 30, 40);
@names = ("John Paul", "Lisa", "Kumar");

print "$ages[0] = $ages[0]\n";
print "$ages[1] = $ages[1]\n";
print "$ages[2] = $ages[2]\n";
print "$names[0] = $names[0]\n";
print "$names[1] = $names[1]\n";
print "$names[2] = $names[2]\n";
```

Here we have used the escape sign (\) before the $ sign just to print it. Other Perl will understand it as a variable and will print its value. When executed, this will produce the following result:

```
$ages[0] = 25
$ages[1] = 30
$ages[2] = 40
$names[0] = John Paul
$names[1] = Lisa
$names[2] = Kumar
```

In Perl, List and Array terms are often used as if they're interchangeable. But the list is the data, and the array is the variable.
Array Creation

Array variables are prefixed with the @ sign and are populated using either parentheses or the qw operator. For example:

```perl
@array = (1, 2, 'Hello');
@array = qw/This is an array/;
```

The second line uses the qw// operator, which returns a list of strings, separating the delimited string by white space. In this example, this leads to a four-element array; the first element is 'this' and last (fourth) is 'array'. This means that you can use different lines as follows:

```perl
@days = qw/Monday
  Tuesday
  ...
  Sunday/;
```

You can also populate an array by assigning each value individually as follows:

```perl
$array[0] = 'Monday';
...
$array[6] = 'Sunday';
```

Accessing Array Elements

When accessing individual elements from an array, you must prefix the variable with a dollar sign ($) and then append the element index within the square brackets after the name of the variable. For example:

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

@days = qw/Mon Tue Wed Thu Fri Sat Sun/;

print "$days[0]\n";
print "$days[1]\n";
print "$days[2]\n";
print "$days[6]\n";
print "$days[-1]\n";
print "$days[-7]\n";
```
This will produce the following result:

| Mon | Tue | Wed | Sun | Sun | Mon |

Array indices start from zero, so to access the first element you need to give 0 as indices. You can also give a negative index, in which case you select the element from the end, rather than the beginning, of the array. This means the following:

```perl
print $days[-1]; # outputs Sun
print $days[-7]; # outputs Mon
```

**Sequential Number Arrays**

Perl offers a shortcut for sequential numbers and letters. Rather than typing out each element when counting to 100 for example, we can do something like as follows:

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

@var_10 = (1..10);
@var_20 = (10..20);
@var_abc = (a..z);

print "@var_10\n"; # Prints number from 1 to 10
print "@var_20\n"; # Prints number from 10 to 20
print "@var_abc\n"; # Prints number from a to z
```

Here double dot (..) is called **range operator**. This will produce the following result:

```
1 2 3 4 5 6 7 8 9 10
10 11 12 13 14 15 16 17 18 19 20
a b c d e f g h i j k l m n o p q r s t u v w x y z
```
**Array Size**

The size of an array can be determined using the scalar context on the array - the returned value will be the number of elements in the array:

```perl
$array = (1,2,3);
print "Size: ", scalar @array,"\n";
```

The value returned will always be the physical size of the array, not the number of valid elements. You can demonstrate this, and the difference between scalar @array and $#array, using this fragment is as follows:

```perl
#!/u/er/bin/perl

$array = (1,2,3);
$array[50] = 4;

,size = @array;
$max_index = $#array;

print "Size: ",$size,"\n";
print "Max Index: ",$max_index,"\n";
```

This will produce the following result:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Size:</td>
<td>51</td>
</tr>
<tr>
<td>Max Index</td>
<td>50</td>
</tr>
</tbody>
</table>

There are only four elements in the array that contains information, but the array is 51 elements long, with a highest index of 50.

**Adding and Removing Elements in Array**

Perl provides a number of useful functions to add and remove elements in an array. You may have a question what is a function? So far you have used `print` function to print various values. Similarly there are various other functions or sometime called sub-routines, which can be used for various other functionalities.

<table>
<thead>
<tr>
<th>S.N.</th>
<th>Types and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. `push @ARRAY, LIST`
   Pushes the values of the list onto the end of the array.

2. `pop @ARRAY`
   Pops off and returns the last value of the array.

3. `shift @ARRAY`
   Shifts the first value of the array off and returns it, shortening the array by 1 and moving everything down.

4. `unshift @ARRAY, LIST`
   Prepends list to the front of the array, and returns the number of elements in the new array.

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

# create a simple array
@coins = ("Quarter","Dime","Nickel");
print "1. \@coins  = @coins\n";

# add one element at the end of the array
push(@coins, "Penny");
print "2. \@coins  = @coins\n";

# add one element at the beginning of the array
unshift(@coins, "Dollar");
print "3. \@coins  = @coins\n";

# remove one element from the last of the array.
pop(@coins);
print "4. \@coins  = @coins\n";

# remove one element from the beginning of the array.
shift(@coins);
```
print "5. @coins = @coins\n";

This will produce the following result:

1. @coins = Quarter Dime Nickel
2. @coins = Quarter Dime Nickel Penny
3. @coins = Dollar Quarter Dime Nickel Penny
4. @coins = Dollar Quarter Dime Nickel
5. @coins = Quarter Dime Nickel

**Slicing Array Elements**

You can also extract a "slice" from an array - that is, you can select more than one item from an array in order to produce another array.

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

@days = qw/Mon Tue Wed Thu Fri Sat Sun/;

@weekdays = @days[3,4,5];

print "@weekdays\n";
```

This will produce the following result:

Thu Fri Sat

The specification for a slice must have a list of valid indices, either positive or negative, each separated by a comma. For speed, you can also use the .. range operator:

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

@days = qw/Mon Tue Wed Thu Fri Sat Sun/;

@weekdays = @days[3..5];

print "@weekdays\n";
```

This will produce the following result:


# Replacing Array Elements

Now we are going to introduce one more function called `splice()`, which has the following syntax:

```perl
splice @ARRAY, OFFSET [ , LENGTH [ , LIST ] ]
```

This function will remove the elements of `@ARRAY` designated by `OFFSET` and `LENGTH`, and replaces them with `LIST`, if specified. Finally, it returns the elements removed from the array. Following is the example:

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

@nums = (1..20);
print "Before - @nums\n";

splice(@nums, 5, 5, 21..25);
print "After - @nums\n";
```

This will produce the following result:

```
Before - 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
After - 1 2 3 4 5 21 22 23 24 25 11 12 13 14 15 16 17 18 19 20
```

Here, the actual replacement begins with the 6th number after that five elements are then replaced from 6 to 10 with the numbers 21, 22, 23, 24 and 25.

## Transform Strings to Arrays

Let's look into one more function called `split()`, which has the following syntax:

```perl
split [ PATTERN [ , EXPR [ , LIMIT ] ] ]
```

This function splits a string into an array of strings, and returns it. If `LIMIT` is specified, splits into at most that number of fields. If `PATTERN` is omitted, splits on whitespace. Following is the example:

```perl
#!/usr/bin/perl
```
# define Strings
$var_string = "Rain-Drops-On-Roses-And-Whiskers-On-Kittens";
$var_names = "Larry,David,Roger,Ken,Michael,Tom"

# transform above strings into arrays.
@string = split('-', $var_string);
@names = split(',', $var_names);

print "$string[3]\n"; # This will print Roses
print "$names[4]\n"; # This will print Michael

This will produce the following result:

Roses
Michael

## Transform Arrays to Strings

We can use the `join()` function to rejoin the array elements and form one long scalar string. This function has the following syntax:

```
join EXPR, LIST
```

This function joins the separate strings of LIST into a single string with fields separated by the value of EXPR, and returns the string. Following is the example:

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

# define Strings
$var_string = "Rain-Drops-On-Roses-And-Whiskers-On-Kittens";
$var_names = "Larry,David,Roger,Ken,Michael,Tom"

# transform above strings into arrays.
@string = split('-', $var_string);
@names = split(',', $var_names);

$string1 = join( '-', @string );
```
$string2 = join( ',', @names );

print "$string1
";
print "$string2
";

This will produce the following result:

Rain-Drops-On-Roses-And-Whiskers-On-Kittens
Larry,David,Roger,Ken,Michael,Tom

**Sorting Arrays**

The `sort()` function sorts each element of an array according to the ASCII Numeric standards. This function has the following syntax:

```
sort [ SUBROUTINE ] LIST
```

This function sorts the LIST and returns the sorted array value. If SUBROUTINE is specified then specified logic inside the SUBROUTINE is applied while sorting the elements.

```
#!/usr/bin/perl

# define an array
@foods = qw(pizza steak chicken burgers);
print "Before: @foods\n";

# sort this array
@foods = sort(@foods);
print "After: @foods\n";
```

This will produce the following result:

Before: pizza steak chicken burgers
After: burgers chicken pizza steak

Please note that sorting is performed based on ASCII Numeric value of the words. So the best option is to first transform every element of the array into lowercase letters and then perform the sort function.
The `$[ Special Variable`

So far you have seen simple variables we defined in our programs and used them to store and print scalar and array values. Perl provides numerous special variables, which have their predefined meaning.

We have a special variable, which is written as `$[`. This special variable is a scalar containing the first index of all arrays. Because Perl arrays have zero-based indexing, `$[` will almost always be 0. But if you set `$[` to 1 then all your arrays will use one-based indexing. It is recommended not to use any other indexing other than zero. However, let’s take one example to show the usage of `$[` variable:

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

# define an array
@foods = qw(pizza steak chicken burgers);
print "Foods: @foods\n";

# Let's reset first index of all the arrays.
$[ = 1;

print "Food at @foods[1]: $foods[1]\n";
print "Food at @foods[2]: $foods[2]\n";
```

This will produce the following result:

<table>
<thead>
<tr>
<th>Foods: pizza steak chicken burgers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Food at @foods[1]: pizza</td>
</tr>
<tr>
<td>Food at @foods[2]: steak</td>
</tr>
</tbody>
</table>

Merging Arrays

Because an array is just a comma-separated sequence of values, you can combine them together as shown below:

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

@numbers = (1,3,(4,5,6));
```
print "numbers = @numbers\n";

This will produce the following result:

numbers = 1 3 4 5 6

The embedded arrays just become a part of the main array as shown below:

#!/usr/bin/perl

@odd = (1, 3, 5);
@even = (2, 4, 6);

@numbers = (@odd, @even);

print "numbers = @numbers\n";

This will produce the following result:

numbers = 1 3 5 2 4 6

### Selecting Elements from Lists

The list notation is identical to that for arrays. You can extract an element from an array by appending square brackets to the list and giving one or more indices:

#!/usr/bin/perl

$var = (5, 4, 3, 2, 1)[4];

print "value of var = $var\n";

This will produce the following result:

value of var = 1

Similarly, we can extract slices, although without the requirement for a leading @ character:

#!/usr/bin/perl
@list = (5,4,3,2,1)[1..3];

print "Value of list = @list\n";

This will produce the following result:

Value of list = 4 3 2
A hash is a set of **key/value** pairs. Hash variables are preceded by a percent (%) sign. To refer to a single element of a hash, you will use the hash variable name preceded by a "$" sign and followed by the "key" associated with the value in curly brackets.

Here is a simple example of using the hash variables:

```perl
#!/usr/bin/perl
%
data = ('John Paul', 45, 'Lisa', 30, 'Kumar', 40);

print "\$data{'John Paul'} = \$data{'John Paul'}\n";
print "\$data{'Lisa'} = \$data{'Lisa'}\n";
print "\$data{'Kumar'} = \$data{'Kumar'}\n";
```

This will produce the following result:

```
$data{'John Paul'} = 45
$data{'Lisa'} = 30
$data{'Kumar'} = 40
```

**Creating Hashes**

Hashes are created in one of the two following ways. In the first method, you assign a value to a named key on a one-by-one basis:

```perl
$data{'John Paul'} = 45;
$data{'Lisa'} = 30;
$data{'Kumar'} = 40;
```

In the second case, you use a list, which is converted by taking individual pairs from the list: the first element of the pair is used as the key, and the second, as the value. For example:

```perl
%data = ('John Paul', 45, 'Lisa', 30, 'Kumar', 40);
```

For clarity, you can use `=>` as an alias for `,` to indicate the key/value pairs as follows:
Here is one more variant of the above form, have a look at it, here all the keys have been preceded by hyphen (-) and no quotation is required around them:

```perl
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
```

But it is important to note that there is a single word, i.e., without spaces keys have been used in this form of hash formation and if you build-up your hash this way then keys will be accessed using hyphen only as shown below.

```perl
$val = %data{-JohnPaul}
$val = %data{-Lisa}
```

### Accessing Hash Elements

When accessing individual elements from a hash, you must prefix the variable with a dollar sign ($) and then append the element key within curly brackets after the name of the variable. For example:

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

%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);

print "$data{'John Paul'}\n";
print "$data{'Lisa'}\n";
print "$data{'Kumar'}\n";
```

This will produce the following result:

```
45
30
40
```

### Extracting Slices

You can extract slices of a hash just as you can extract slices from an array. You will need to use @ prefix for the variable to store the returned value because they will be a list of values:

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

```perl
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);

@array = @data{ 'John Paul', 'Lisa' };

print "Array : @array\n";

This will produce the following result:

Array : 45 30
```

### Extracting Keys and Values

You can get a list of all of the keys from a hash by using `keys` function, which has the following syntax:

```perl
keys %HASH
```

This function returns an array of all the keys of the named hash. Following is the example:

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

%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);

@names = keys %data;

print "$names[0]\n";
print "$names[1]\n";
print "$names[2]\n";

This will produce the following result:

Lisa
John Paul
Kumar
```

Similarly, you can use `values` function to get a list of all the values. This function has the following syntax:
This function returns a normal array consisting of all the values of the named hash. Following is the example:

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

%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);

@ages = values %data;

print "$ages[0]\n";
print "$ages[1]\n";
print "$ages[2]\n";
```

This will produce the following result:

```
30
45
40
```

### Checking for Existence

If you try to access a key/value pair from a hash that doesn't exist, you'll normally get the *undefined* value, and if you have warnings switched on, then you'll get a warning generated at run time. You can get around this by using the `exists` function, which returns true if the named key exists, irrespective of what its value might be:

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

%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);

if( exists($data{'Lisa'}) ){
    print "Lisa is $data{'Lisa'} years old\n";
}
else{
    print "I don't know age of Lisa\n";
```
Here we have introduced the IF...ELSE statement, which we will study in a separate chapter. For now you just assume that \texttt{if(condition)} part will be executed only when the given condition is true otherwise \texttt{else} part will be executed. So when we execute the above program, it produces the following result because here the given condition \texttt{exists($data{'Lisa'})} returns true:

\begin{verbatim}
Lisa is 30 years old
\end{verbatim}

\section*{Getting Hash Size}

You can get the size - that is, the number of elements from a hash by using the scalar context on either keys or values. Simply saying first you have to get an array of either the keys or values and then you can get the size of array as follows:

\begin{verbatim}
#!/usr/bin/perl

%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);

@keys = keys %data;
 dataSize = @keys;
print "1 - Hash size: is $size\n";

@values = values %data;
$dataSize = @values;
print "2 - Hash size: is $size\n";
\end{verbatim}

This will produce the following result:

\begin{verbatim}
1 - Hash size: is 3
2 - Hash size: is 3
\end{verbatim}

\section*{Add and Remove Elements in Hashes}

Adding a new key/value pair can be done with one line of code using simple assignment operator. But to remove an element from the hash you need to use \texttt{delete} function as shown below in the example:

\begin{verbatim}
#!/usr/bin/perl

%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);

delete %data;

@keys = keys %data;
$dataSize = @keys;
print "1 - Hash size: is $size\n";

@values = values %data;
$dataSize = @values;
print "2 - Hash size: is $size\n";
\end{verbatim}

This will produce the following result:

\begin{verbatim}
1 - Hash size: is 0
2 - Hash size: is 0
\end{verbatim}
```perl
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
@keys = keys %data;
,size = @keys;
print "1 - Hash size: is $size\n";

# adding an element to the hash;
$data{'Ali'} = 55;
@keys = keys %data;
,size = @keys;
print "2 - Hash size: is $size\n";

# delete the same element from the hash;
delete $data{'Ali'};
@keys = keys %data;
,size = @keys;
print "3 - Hash size: is $size\n";
```

This will produce the following result:

1 - Hash size: is 3
2 - Hash size: is 4
3 - Hash size: is 3
Perl conditional statements helps in the decision making, which require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages:

The number 0, the strings '0' and "", the empty list (), and undef are all **false** in a boolean context and all other values are **true**. Negation of a true value by ! or not returns a special false value.

Perl programming language provides the following types of conditional statements.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>if statement</td>
<td>An <em>if</em> statement consists of a boolean expression followed by one or more statements.</td>
</tr>
<tr>
<td>if...else statement</td>
<td>An <em>if</em> statement can be followed by an }</td>
</tr>
</tbody>
</table>

49
if...elsif...else statement

An if statement can be followed by an optional elsif statement and then by an optional else statement.

optional else statement.

unless statement

An unless statement consists of a boolean expression followed by one or more statements.

unless...else statement

An unless statement can be followed by an optional else statement.

unless...elsif..else statement

An unless statement can be followed by an optional elsif statement and then by an optional else statement.

switch statement

With the latest versions of Perl, you can make use of the switch statement, which allows a simple way of comparing a variable value against various conditions.

if statement

A Perl if statement consists of a boolean expression followed by one or more statements.

Syntax

The syntax of an if statement in Perl programming language is:

```perl
if(boolean_expression){
    # statement(s) will execute if the given condition is true
}
```

If the boolean expression evaluates to true then the block of code inside the if statement will be executed. If boolean expression evaluates to false then the first set of code after the end of the if statement (after the closing curly brace) will be executed.

The number 0, the strings '0' and "", the empty list () , and undef are all false in a boolean context and all other values are true. Negation of a true value by ! or not returns a special false value.
#!/usr/local/bin/perl

$a = 10;
# check the boolean condition using if statement
if( $a < 20 ){
    # if condition is true then print the following
    printf "a is less than 20\n";
}
print "value of a is : $a\n";

$a = "";
# check the boolean condition using if statement
if( $a ){
    # if condition is true then print the following
    printf "a has a true value\n";
}
print "value of a is : $a\n";
First IF statement makes use of less than operator (<), which compares two operands and if first operand is less than the second one then it returns true otherwise it returns false. So when the above code is executed, it produces the following result:

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a is less than 20</td>
<td></td>
</tr>
<tr>
<td>value of a is : 10</td>
<td></td>
</tr>
<tr>
<td>value of a is :</td>
<td></td>
</tr>
</tbody>
</table>

**if...else statement**

A Perl if statement can be followed by an optional else statement, which executes when the boolean expression is false.

**Syntax**

The syntax of an **if...else** statement in Perl programming language is:

```perl
if(boolean_expression){
    # statement(s) will execute if the given condition is true
}else{
    # statement(s) will execute if the given condition is false
}
```

If the boolean expression evaluates to **true**, then the if block of code will be executed otherwise else block of code will be executed.

The number 0, the strings '0' and '', the empty list (), and undef are all **false** in a boolean context and all other values are **true**. Negation of a true value by ! or not returns a special false value.
Example

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

$a = 100;
# check the boolean condition using if statement
if( $a < 20 ){
    # if condition is true then print the following
    printf "a is less than 20\n";
}else{
    # if condition is false then print the following
    printf "a is greater than 20\n";
}
print "value of a is : $a\n";

$a = "";
# check the boolean condition using if statement
if( $a ){
    # if condition is true then print the following
    printf "a has a true value\n";
}
```
When the above code is executed, it produces the following result:

```
a is greater than 20
value of a is : 100
a has a false value
value of a is :  
```

### if...elsif...else statement

An `if` statement can be followed by an optional `elsif...else` statement, which is very useful to test the various conditions using single `if...elsif` statement.

When using `if`, `elsif`, `else` statements there are few points to keep in mind.

- An `if` can have zero or one `else`'s and it must come after any `elsif`'s.
- An `if` can have zero to many `elsif`'s and they must come before the `else`.
- Once an `elsif` succeeds, none of the remaining `elsif`'s or `else`'s will be tested.

### Syntax

The syntax of an `if...elsif...else` statement in Perl programming language is:

```perl
if(boolean_expression 1){
    # Executes when the boolean expression 1 is true
}
elsif( boolean_expression 2){
    # Executes when the boolean expression 2 is true
}
elsif( boolean_expression 3){
    # Executes when the boolean expression 3 is true
}
else{
    # If none of the above conditions are met
}
```
# Executes when none of the above condition is true
}

Example

#!/usr/local/bin/perl

$a = 100;
# check the boolean condition using if statement
if( $a == 20 ){
    # if condition is true then print the following
    printf "a has a value which is 20\n";
}elsif( $a == 30 ){
    # if condition is true then print the following
    printf "a has a value which is 30\n";
}else{
    # if none of the above conditions is true
    printf "a has a value which is $a\n";
}

Here we are using the equality operator == which is used to check if two operands are equal or not. If both the operands are same, then it returns true otherwise it returns false. When the above code is executed, it produces the following result:

a has a value which is 100

unless statement

A Perl unless statement consists of a boolean expression followed by one or more statements.

Syntax

The syntax of an unless statement in Perl programming language is:

unless(boolean_expression){
    # statement(s) will execute if the given condition is false
}

55
If the boolean expression evaluates to \textbf{false}, then the block of code inside the unless statement will be executed. If boolean expression evaluates to \textbf{true} then the first set of code after the end of the unless statement (after the closing curly brace) will be executed.

The number 0, the strings '0' and "", the empty list (), and undef are all \textbf{false} in a boolean context and all other values are \textbf{true}. Negation of a true value by ! or \textbf{not} returns a special false value.

\textbf{Flow Diagram}

\textbf{Example}

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

$a = 20;
# check the boolean condition using unless statement
unless( $a < 20 ){  
    # if condition is false then print the following
    printf "a is not less than 20\n";
}
print "value of a is : $a\n";

$a = "";
# check the boolean condition using unless statement
```
unless ( $a ){
    # if condition is false then print the following
    printf "a has a false value\n";
}
print "value of a is : $a\n";

First unless statement makes use of less than operator (<), which compares two operands and if first operand is less than the second one then it returns true otherwise it returns false. So when the above code is executed, it produces the following result:

```
a is not less than 20
value of a is : 20
a has a false value
value of a is :
```

**unless...else statement**

A Perl `unless` statement can be followed by an optional `else` statement, which executes when the boolean expression is true.

**Syntax:**

The syntax of an `unless...else` statement in Perl programming language is:

```
unless(boolean_expression){
    # statement(s) will execute if the given condition is false
}else{
    # statement(s) will execute if the given condition is true
}
```

If the boolean expression evaluates to `true` then the `unless block` of code will be executed otherwise `else block` of code will be executed.

The number 0, the strings '0' and "" , the empty list () , and undef are all `false` in a boolean context and all other values are `true`. Negation of a true value by ! or `not` returns a special false value.
Example

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

$a = 100;
# check the boolean condition using unless statement
unless( $a == 20 ){  
   # if condition is false then print the following
   printf "given condition is false\n";
}else{
   # if condition is true then print the following
   printf "given condition is true\n";
}
print "value of a is : $a\n";

$a = "";
# check the boolean condition using unless statement
unless( $a ){  
   # if condition is false then print the following
   printf "a has a false value\n";
```
else{
    # if condition is true then print the following
    printf "a has a true value\n";
}
print "value of a is : $a\n";

When the above code is executed, it produces the following result:

given condition is false
value of a is : 100
a has a false value
value of a is :

unless...elsif..else statement

An unless statement can be followed by an optional elsif...else statement, which is very useful to test the various conditions using single unless...elsif statement.

When using unless, elsif, else statements there are few points to keep in mind.

- An unless can have zero or one else's and it must come after any elsif's.
- An unless can have zero to many elsif's and they must come before the else.
- Once an elsif succeeds, none of the remaining elsif's or else's will be tested.

Syntax

The syntax of an unless...elsif...else statement in Perl programming language is:

unless(boolean_expression 1){
    # Executes when the boolean expression 1 is false
}
elsif( boolean_expression 2){
    # Executes when the boolean expression 2 is true
}
elsif( boolean_expression 3){
    # Executes when the boolean expression 3 is true

Example

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

$a = 20;
# check the boolean condition using if statement
unless( $a == 30 ){
    # if condition is false then print the following
    printf "a has a value which is not 20\n";
}elsif( $a == 30 ){
    # if condition is true then print the following
    printf "a has a value which is 30\n";
}else{
    # if none of the above conditions is met
    printf "a has a value which is $a\n";
}
```

Here we are using the equality operator == which is used to check if two operands are equal or not. If both the operands are same then it returns true, otherwise it returns false. When the above code is executed, it produces the following result:

```
a has a value which is not 20
```

**switch statement**

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

A switch case implementation is dependent on **Switch** module and **Switch** module has been implemented using **Filter::Util::Call** and **Text::Balanced** and requires both these modules to be installed.
Syntax

The synopsis for a **switch** statement in Perl programming language is as follows:

```perl
use Switch;

switch(argument){
    case 1         { print "number 1" }
    case "a"       { print "string a" }
    case [1..10,42] { print "number in list" }
    case (@array)  { print "number in list" }
    case /\w+/     { print "pattern" }
    case qr/\w+/   { print "pattern" }
    case (%hash)   { print "entry in hash" }
    case (&sub)    { print "arg to subroutine" }
    else           { print "previous case not true" }
}
```

The following rules apply to a **switch** statement:

- The **switch** statement takes a single scalar argument of any type, specified in parentheses.
- The value is followed by a block, which may contain one or more case statements followed by a block of Perl statement(s).
- A case statement takes a single scalar argument and selects the appropriate type of matching between the case argument and the current switch value.
- If the match is successful, the mandatory block associated with the case statement is executed.
- A **switch** statement can have an optional **else** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is matched.
- If a case block executes an untargeted **next**, control is immediately transferred to the statement after the case statement (i.e., usually another case), rather than out of the surrounding switch block.
- Not every case needs to contain a **next**. If no **next** appears, the flow of control will not fall through subsequent cases.
Flow Diagram

Example

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

use Switch;

$var = 10;
$array = (10, 20, 30);
%hash = ('key1' => 10, 'key2' => 20);

switch($var){
    case 10 { print "number 100\n" }
    case "a" { print "string a" }
    case [1..10,42] { print "number in list" }
    case (@array) { print "number in list" }
    case (%hash) { print "entry in hash" }
    else { print "previous case not true" }
}
```
When the above code is executed, it produces the following result:

```
number 100
```

Fall-through is usually a bad idea in a switch statement. However, now consider a fall-through case, we will use the `next` to transfer the control to the next matching case, which is a list in this case:

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

use Switch;

$var = 10;
$array = (10, 20, 30);
$hash = ('key1' => 10, 'key2' => 20);

switch($var){
    case 10  { print "number 100\n"; next; }
    case "a" { print "string a" }
    case [1..10,42] { print "number in list" }
    case (@array) { print "number in list" }
    case (%hash)  { print "entry in hash" }
    else         { print "previous case not true" }
}
```

When the above code is executed, it produces the following result:

```
number 100
number in list
```

**The ? : Operator**

Let's check the `conditional operator` `?:` which can be used to replace `if...else` statements. It has the following general form:

```
Exp1 ? Exp2 : Exp3;
```
Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a ? expression is determined like this: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression. Below is a simple example making use of this operator:

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

$name = "Ali";
$age = 10;

$status = ($age > 60 ) ? "A senior citizen" : "Not a senior citizen";

print "$name is - $status\n";
```

This will produce the following result:

Ali is - Not a senior citizen
There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:

Perl programming language provides the following types of loop to handle the looping requirements.

<table>
<thead>
<tr>
<th>Loop Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>while loop</td>
<td>Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.</td>
</tr>
<tr>
<td>until loop</td>
<td>Repeats a statement or group of statements until a given condition becomes true. It tests the condition</td>
</tr>
</tbody>
</table>
before executing the loop body.

<table>
<thead>
<tr>
<th>for loop</th>
<th>Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.</th>
</tr>
</thead>
<tbody>
<tr>
<td>foreach loop</td>
<td>The foreach loop iterates over a normal list value and sets the variable VAR to be each element of the list in turn.</td>
</tr>
<tr>
<td>do...while loop</td>
<td>Like a while statement, except that it tests the condition at the end of the loop body</td>
</tr>
<tr>
<td>nested loops</td>
<td>You can use one or more loop inside any another while, for or do..while loop.</td>
</tr>
</tbody>
</table>

**while loop**

A **while** loop statement in Perl programming language repeatedly executes a target statement as long as a given condition is true.

**Syntax**

The syntax of a **while** loop in Perl programming language is:

```perl
while(condition) {
    statement(s);
}
```

Here **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.

The number 0, the strings '0' and "", the empty list (), and undef are all **false** in a boolean context and all other values are **true**. Negation of a true value by ! or **not** returns a special false value.
Here the key point of the `while` loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

**Example**

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

$a = 10;

# while loop execution
while( $a < 20 ){
    printf "Value of a: $a\n";
    $a = $a + 1;
}
```

Here we are using the comparison operator `<` to compare value of variable `$a` against 20. So while value of `$a` is less than 20, `while` loop continues executing a block of code next to it and as soon as the value of `$a` becomes equal to 20, it comes out. When executed, above code produces the following result:
An **until** loop statement in Perl programming language repeatedly executes a target statement as long as a given condition is false.

**Syntax**

The syntax of an **until** loop in Perl programming language is:

```perl
until(condition)
{
    statement(s);
}
```

Here **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression. The loop iterates until the condition becomes true. When the condition becomes true, the program control passes to the line immediately following the loop.

The number 0, the strings '0' and '', the empty list (), and undef are all **false** in a boolean context and all other values are **true**. Negation of a true value by ! or **not** returns a special false value.
Here key point of the *until* loop is that the loop might not ever run. When the condition is tested and the result is true, the loop body will be skipped and the first statement after the until loop will be executed.

**Example**

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

$a = 5;

# until loop execution
until( $a > 10 ){  
    printf "Value of a: $a\n";
    $a = $a + 1;
}
```

Here we are using the comparison operator > to compare value of variable $a against 10. So until the value of $a is less than 10, *until* loop continues executing a block of code next to it and as soon as the value of $a becomes
greater than 10, it comes out. When executed, above code produces the following result:

<table>
<thead>
<tr>
<th>Value of a:</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of a:</td>
<td>6</td>
</tr>
<tr>
<td>Value of a:</td>
<td>7</td>
</tr>
<tr>
<td>Value of a:</td>
<td>8</td>
</tr>
<tr>
<td>Value of a:</td>
<td>9</td>
</tr>
<tr>
<td>Value of a:</td>
<td>10</td>
</tr>
</tbody>
</table>

**for loop**

A *for* loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

**Syntax**

The syntax of a *for* loop in Perl programming language is:

```perl
for ( init; condition; increment ){
    statement(s);
}
```

Here is the flow of control in a *for* loop:

1. The *init* step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

2. Next, the *condition* is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.

3. After the body of the for loop executes, the flow of control jumps back up to the *increment* statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.

4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.
#!/usr/local/bin/perl

# for loop execution
for( $a = 10; $a < 20; $a = $a + 1 ){
    print "value of a: $a\n";
}

When the above code is executed, it produces the following result:

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

**foreach loop**

The **foreach** loop iterates over a list value and sets the control variable (var) to be each element of the list in turn:

**Syntax**

The syntax of a **foreach** loop in Perl programming language is:

```perl
foreach var (list) {
    ...
}
```
Flow Diagram

Example

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

@list = (2, 20, 30, 40, 50);

# foreach loop execution
foreach $a (@list){
    print "value of a: $a\n";
}
```

When the above code is executed, it produces the following result:

```
value of a: 2
value of a: 20
value of a: 30
value of a: 40
value of a: 50
```
do...while loop

Unlike for and while loops, which test the loop condition at the top of the loop, the do...while loop checks its condition at the bottom of the loop.

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax

The syntax of a do...while loop in Perl is:

```
do
{
    statement(s);
}while( condition );
```

It should be noted that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested. If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

The number 0, the strings '0' and "" , the empty list () , and undef are all false in a boolean context and all other values are true. Negation of a true value by ! or not returns a special false value.

Flow Diagram
Example

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

$a = 10;

# do...while loop execution
do{
    printf "Value of a: $a\n";
    $a = $a + 1;
}while( $a < 20 );
```

When the above code is executed, it produces the following result:

| Value of a: 10 |
| Value of a: 11 |
| Value of a: 12 |
| Value of a: 13 |
| Value of a: 14 |
| Value of a: 15 |
| Value of a: 16 |
| Value of a: 17 |
| Value of a: 18 |
| Value of a: 19 |

**nested loops**

A loop can be nested inside of another loop. Perl allows to nest all type of loops to be nested.

**Syntax**

The syntax for a **nested for loop** statement in Perl is as follows:

```perl
for ( init; condition; increment ){
    for ( init; condition; increment ){
        statement(s);
    }
}
```
The syntax for a nested while loop statement in Perl is as follows:

```perl
while(condition){
    while(condition){
        statement(s);
    }
    statement(s);
}
```

The syntax for a nested do...while loop statement in Perl is as follows:

```perl
do{
    statement(s);
    do{
        statement(s);
    }while( condition );
}while( condition );
```

The syntax for a nested until loop statement in Perl is as follows:

```perl
until(condition){
    until(condition){
        statement(s);
    }
    statement(s);
}
```

The syntax for a nested foreach loop statement in Perl is as follows:

```perl
foreach $a (@listA){
    foreach $b (@listB){
        statement(s);
    }
    statement(s);
}
```
Example
The following program uses a nested \texttt{while} loop to show the usage:

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

$a = 0;
$b = 0;

# outer while loop
while($a < 3){
    $b = 0;
    # inner while loop
    while( $b < 3 ){
        print "value of a = $a, b = $b\n";
        $b = $b + 1;
    }
    $a = $a + 1;
    print "Value of a = $a\n\n";
}
```

This would produce the following result:

- value of a = 0, b = 0
- value of a = 0, b = 1
- value of a = 0, b = 2
- Value of a = 1

- value of a = 1, b = 0
- value of a = 1, b = 1
- value of a = 1, b = 2
- Value of a = 2

- value of a = 2, b = 0
- value of a = 2, b = 1
value of a = 2, b = 2
Value of a = 3

**Loop Control Statements**

Loop control statements change the execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C supports the following control statements. Click the following links to check their detail.

<table>
<thead>
<tr>
<th>Control Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>next statement</td>
<td>Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.</td>
</tr>
<tr>
<td>last statement</td>
<td>Terminates the loop statement and transfers execution to the statement immediately following the loop.</td>
</tr>
<tr>
<td>continue statement</td>
<td>A continue BLOCK, it is always executed just before the conditional is about to be evaluated again.</td>
</tr>
<tr>
<td>redo statement</td>
<td>The redo command restarted the loop block without evaluating the conditional again. The continue block, if any, is not executed.</td>
</tr>
<tr>
<td>goto statement</td>
<td>Perl supports a goto command with three forms: goto label, goto expr, and goto &amp;name.</td>
</tr>
</tbody>
</table>

**next statement**

The Perl `next` statement starts the next iteration of the loop. You can provide a LABEL with `next` statement where LABEL is the label for a loop. A `next` statement can be used inside a nested loop where it will be applicable to the nearest loop if a LABEL is not specified.

If there is a `continue` block on the loop, it is always executed just before the condition is about to be evaluated. You will see the continue statement in separate chapter.
Syntax

The syntax of a `next` statement in Perl is:

```
next [ LABEL ];
```

A LABEL inside the square braces indicates that LABEL is optional and if a LABEL is not specified, then next statement will jump the control to the next iteration of the nearest loop.

Flow Diagram

Example

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

$a = 10;
while( $a < 20 ){
    if( $a == 15 )
        {
            # skip the iteration.
            $a = $a + 1;
            next;
        }
    print "value of a: $a\n";
}
$a = $a + 1;
}

When the above code is executed, it produces the following result:

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19

Let's take one example where we are going to use a LABEL along with next statement:

#!/usr/local/bin/perl

$a = 0;
OUTER: while ( $a < 4 ){
    $b = 0;
    print "value of a: $a\n";
    INNER:while ( $b < 4){
        if( $a == 2){
            $a = $a + 1;
            # jump to outer loop
            next OUTER;
        }
        $b = $b + 1;
        print "Value of b : $b\n";
    }
    $b = $b + 1;
    print "Value of b : $b\n";
}
print "\n";
@a = @a + 1;
When the above code is executed, it produces the following result:

<table>
<thead>
<tr>
<th>value of a</th>
<th>Value of b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

last statement

When a last statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop. You can provide a LABEL with last statement where LABEL is the label for a loop. A last statement can be used inside a nested loop where it will be applicable to the nearest loop if a LABEL is not specified.

If there is any continue block on the loop, then it is not executed. You will see the continue statement in a separate chapter.

Syntax

The syntax of a last statement in Perl is:

```
last [LABEL];
```
Flow Diagram

Example 1

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

$a = 10;
while( $a < 20 ){
    if( $a == 15 ){
        # terminate the loop.
        $a = $a + 1;
    last;
    }
    print "value of a: $a\n";
    $a = $a + 1;
}

When the above code is executed, it produces the following result:

value of a: 10
value of a: 11
value of a: 12
```
Example 2
Let's take one example where we are going to use a LABEL along with next statement:

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

$a = 0;

OUTER: while( $a < 4 ){ 
    $b = 0;
    print "value of a: $a\n";
    INNER:while ( $b < 4){
        if( $a == 2){
            # terminate outer loop
            last OUTER;
        }
        $b = $b + 1;
        print "Value of b : $b\n";
    }
    $b = $b + 1;
    print "\n";
    $a = $a + 1;
}
```

When the above code is executed, it produces the following result:

```
value of a: 0
Value of b : 1
Value of b : 2
Value of b : 3
Value of b : 4
value of a: 1
Value of b : 1
Value of b : 2
```
continue statement

A continue BLOCK, is always executed just before the conditional is about to be evaluated again. A continue statement can be used with while and foreach loops. A continue statement can also be used alone along with a BLOCK of code in which case it will be assumed as a flow control statement rather than a function.

Syntax

The syntax for a continue statement with while loop is as follows:

```
while(condition){
    statement(s);
}continue{
    statement(s);
}
```

The syntax for a continue statement with foreach loop is as follows:

```
foreach $a (@listA){
    statement(s);
}continue{
    statement(s);
}
```

The syntax for a continue statement with a BLOCK of code is as follows:

```
continue{
    statement(s);
}
```

Example

The following program simulates a for loop using a while loop:

```
#/usr/local/bin/perl
```
$a = 0;
while($a < 3){
    print "Value of a = $a\n";
}continue{
    $a = $a + 1;
}

This would produce the following result:

Value of a = 0
Value of a = 1
Value of a = 2

The following program shows the usage of `continue` statement with `foreach` loop:

```
#/usr/local/bin/perl

@list = (1, 2, 3, 4, 5);
foreach $a (@list){
    print "Value of a = $a\n";
}continue{
    last if $a == 4;
}
```

This would produce the following result:

Value of a = 1
Value of a = 2
Value of a = 3
Value of a = 4

**redo statement**

The `redo` command restarts the loop block without evaluating the conditional again. You can provide a LABEL with `redo` statement where LABEL is the label for a loop. A `redo` statement can be used inside a nested loop where it will be applicable to the nearest loop if a LABEL is not specified.
If there is any `continue` block on the loop, then it will not be executed before evaluating the condition.

**Syntax**
The syntax for a `redo` statement is as follows:

```
redo [LABEL]
```

**Flow Diagram**

```
# Flow Diagram of redo statement
```

**Example**
The following program shows the usage of `redo` statement:

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

$a = 0;
while($a < 10){
    if( $a == 5 ){
        $a = $a + 1;
        redo;
    }
    print "Value of a = $a\n";
}continue{
```
```perl
$a = $a + 1;
}
```

### goto statement

Perl does support a `goto` statement. There are three forms: `goto LABEL`, `goto EXPR`, and `goto &NAME`.

<table>
<thead>
<tr>
<th>S.N.</th>
<th>goto type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>goto LABEL</code></td>
<td>The <code>goto LABEL</code> form jumps to the statement labeled with <code>LABEL</code> and resumes execution from there.</td>
</tr>
<tr>
<td>2</td>
<td><code>goto EXPR</code></td>
<td>The <code>goto EXPR</code> form is just a generalization of <code>goto LABEL</code>. It expects the expression to return a label name and then jumps to that labeled statement.</td>
</tr>
<tr>
<td>3</td>
<td><code>goto &amp;NAME</code></td>
<td>It substitutes a call to the named subroutine for the currently running subroutine.</td>
</tr>
</tbody>
</table>

### Syntax

The syntax for a `goto` statements is as follows:

```perl
goto LABEL
or
goto EXPR
or
goto &NAME
```
Example
The following program shows the most frequently used form of goto statement:

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

$a = 10;

LOOP:do
{
   if ($a == 15){
      # skip the iteration.
      $a = $a + 1;
      # use goto LABEL form
      goto LOOP;
   }
   print "Value of a = $a\n";
   $a = $a + 1;
}while( $a < 20 );
```

When the above code is executed, it produces the following result:

```
Value of a = 10
```
Following example shows the usage of goto EXPR form. Here we are using two strings and then concatenating them using string concatenation operator (.). Finally, its forming a label and goto is being used to jump to the label:

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

$a = 10;
$str1 = "LO";
$str2 = "OP";

LOOP:do
{
    if( $a == 15){
        # skip the iteration.
        $a = $a + 1;
        # use goto EXPR form
        goto $str1.$str2;
    }
    print "Value of a = $a\n";
    $a = $a + 1;
}while( $a < 20 );
```

When the above code is executed, it produces the following result:

```
Value of a = 10
Value of a = 11
Value of a = 12
```

Value of a = 11
Value of a = 12
Value of a = 13
Value of a = 14
Value of a = 16
Value of a = 17
Value of a = 18
Value of a = 19
The Infinite Loop

A loop becomes infinite loop if a condition never becomes false. The for loop is traditionally used for this purpose. Since none of the three expressions that form the for loop are required, you can make an endless loop by leaving the conditional expression empty.

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

for( ; ; )
{
    printf "This loop will run forever.\n";
}
```

You can terminate the above infinite loop by pressing the Ctrl + C keys.

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but as a programmer more commonly use the for ( ; ; ) construct to signify an infinite loop.
What is an Operator?

Simple answer can be given using the expression $4 + 5$ is equal to 9. Here 4 and 5 are called operands and + is called operator. Perl language supports many operator types, but following is a list of important and most frequently used operators:

- Arithmetic Operators
- Equality Operators
- Logical Operators
- Assignment Operators
- Bitwise Operators
- Logical Operators
- Quote-like Operators
- Miscellaneous Operators

Let's have a look at all the operators one by one.

Perl Arithmetic Operators

Assume variable $a holds 10 and variable $b holds 20, then following are the Perl arithmetic operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition - Adds values on either side of the operator</td>
<td>$a + b$ will give 30</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction - Subtracts right hand operand from left hand operand</td>
<td>$a - b$ will give -10</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication - Multiplies values on either side of the operator</td>
<td>$a * b$ will give 200</td>
</tr>
<tr>
<td>Operator</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td><code>/</code></td>
<td>Division - Divides left hand operand by right hand operand</td>
<td><code>$b / $a</code> will give 2</td>
</tr>
<tr>
<td><code>%</code></td>
<td>Modulus - Divides left hand operand by right hand operand and returns remainder</td>
<td><code>$b % $a</code> will give 0</td>
</tr>
<tr>
<td><code>**</code></td>
<td>Exponent - Performs exponential (power) calculation on operators</td>
<td><code>$a**$b</code> will give 10 to the power 20</td>
</tr>
</tbody>
</table>

**Example**

Try the following example to understand all the arithmatic operators available in Perl. Copy and paste the following Perl program in test.pl file and execute this program.

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

$a = 21;
$b = 10;

print "Value of \$a = \$a and value of \$b = \$b\n";

$c = $a + $b;
print 'Value of $a + $b = ' . $c . "\n";

$c = $a - $b;
print 'Value of $a - $b = ' . $c . "\n";

$c = $a * $b;
print 'Value of $a * $b = ' . $c . "\n";

$c = $a / $b;
print 'Value of $a / $b = ' . $c . "\n";

$c = $a % $b;
print 'Value of $a % $b = ' . $c . "\n";
```
$a = 2;
$b = 4;
$c = $a ** $b;
print 'Value of $a ** $b = ' . $c . "\n";

When the above code is executed, it produces the following result:

Value of $a = 21 and value of $b = 10
Value of $a + $b = 31
Value of $a - $b = 11
Value of $a * $b = 210
Value of $a / $b = 2.1
Value of $a % $b = 1
Value of $a ** $b = 16

Perl Equality Operators

These are also called relational operators. Assume variable $a holds 10 and variable $b holds 20 then, lets check the following numeric equality operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Checks if the value of two operands are equal or not, if yes then condition becomes true.</td>
<td>($a == $b) is not true.</td>
</tr>
<tr>
<td>!=</td>
<td>Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.</td>
<td>($a != $b) is true.</td>
</tr>
<tr>
<td>&lt;=&gt;</td>
<td>Checks if the value of two operands are equal or not, and returns -1, 0, or 1 depending on whether the left argument is numerically less than, equal to, or greater than the right argument.</td>
<td>($a &lt;=&gt; $b) returns -1.</td>
</tr>
<tr>
<td>&gt;</td>
<td>Checks if the value of left operand is greater than the value of right operand,</td>
<td>($a &gt; $b) is not true.</td>
</tr>
</tbody>
</table>
if yes then condition becomes true.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.</td>
<td>($a &lt; b) is true.</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.</td>
<td>($a &gt;= b) is not true.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.</td>
<td>($a &lt;= b) is true.</td>
</tr>
</tbody>
</table>

**Example**

Try the following example to understand all the numeric equality operators available in Perl. Copy and paste the following Perl program in test.pl file and execute this program.

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

$a = 21;
$b = 10;

print "Value of \$a = $a and value of \$b = $b\n";

if( $a == $b ){
    print "$a == \$b is true\n";
}else{
    print "$a == \$b is not true\n";
}

if( $a != $b ){
    print "$a != \$b is true\n";
}else{
    print "$a != \$b is not true\n";
}
```
When the above code is executed, it produces the following result:

Value of $a = 21 and value of $b = 10
$a == $b is not true
$a != $b is true
$a <=> $b returns 1
Below is a list of equity operators. Assume variable \$a holds "abc" and variable \$b holds "xyz" then, lets check the following string equality operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>lt</td>
<td>Returns true if the left argument is stringwise less than the right argument.</td>
<td>($a \lt $b) is true.</td>
</tr>
<tr>
<td>gt</td>
<td>Returns true if the left argument is stringwise greater than the right argument.</td>
<td>($a \gt $b) is false.</td>
</tr>
<tr>
<td>le</td>
<td>Returns true if the left argument is stringwise less than or equal to the right argument.</td>
<td>($a \le $b) is true.</td>
</tr>
<tr>
<td>ge</td>
<td>Returns true if the left argument is stringwise greater than or equal to the right argument.</td>
<td>($a \ge $b) is false.</td>
</tr>
<tr>
<td>eq</td>
<td>Returns true if the left argument is stringwise equal to the right argument.</td>
<td>($a \eq $b) is false.</td>
</tr>
<tr>
<td>ne</td>
<td>Returns true if the left argument is stringwise not equal to the right argument.</td>
<td>($a \ne $b) is true.</td>
</tr>
<tr>
<td>cmp</td>
<td>Returns -1, 0, or 1 depending on whether the left argument is stringwise less than, equal to, or greater than the right argument.</td>
<td>($a \cmp $b) is -1.</td>
</tr>
</tbody>
</table>

**Example**

Try the following example to understand all the string equality operators available in Perl. Copy and paste the following Perl program in test.pl file and execute this program.

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

\$a = "abc";
```
$b = "xyz";

print "Value of \$a = \$a and value of \$b = \$b\n";

if( $a lt $b ){
    print "$a lt \$b is true\n";
}else{
    print "$a lt \$b is not true\n";
}

if( $a gt $b ){
    print "$a gt \$b is true\n";
}else{
    print "$a gt \$b is not true\n";
}

if( $a le $b ){
    print "$a le \$b is true\n";
}else{
    print "$a le \$b is not true\n";
}

if( $a ge $b ){
    print "$a ge \$b is true\n";
}else{
    print "$a ge \$b is not true\n";
}

if( $a ne $b ){
    print "$a ne \$b is true\n";
}else{
    print "$a ne \$b is not true\n";
Perl

```perl
$c = $a cmp $b;
print "\$a cmp \$b returns \$c\n";
```

When the above code is executed, it produces the following result:

Value of $a = abc and value of $b = xyz
abc lt $b is true
$a gt $b is not true
$a le $b is true
$a ge $b is not true
$a ne $b is true
$a cmp $b returns -1

### Perl Assignment Operators

Assume variable $a holds 10 and variable $b holds 20, then below are the assignment operators available in Perl and their usage:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Simple assignment operator, Assigns values from right side operands to left side operand</td>
<td>$c = $a + $b will assigned value of $a + $b into $c</td>
</tr>
<tr>
<td>+=</td>
<td>Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand</td>
<td>$c += $a is equivalent to $c = $c + $a</td>
</tr>
<tr>
<td>-=</td>
<td>Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand</td>
<td>$c -= $a is equivalent to $c = $c - $a</td>
</tr>
<tr>
<td>*=</td>
<td>Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand</td>
<td>$c *= $a is equivalent to $c = $c * $a</td>
</tr>
<tr>
<td>Operator</td>
<td>Description</td>
<td>Equivalent Expression</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>/=</td>
<td>Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand</td>
<td>$c /= a$ is equivalent to $c = c / a$</td>
</tr>
<tr>
<td>%=</td>
<td>Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand</td>
<td>$c %= a$ is equivalent to $c = c % a$</td>
</tr>
<tr>
<td>**=</td>
<td>Exponent AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand</td>
<td>$c **= a$ is equivalent to $c = c ** a$</td>
</tr>
</tbody>
</table>

**Example**

Try the following example to understand all the assignment operators available in Perl. Copy and paste the following Perl program in test.pl file and execute this program.

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

$a = 10;
$b = 20;

print "Value of \$a = $a and value of \$b = $b\n";

$c = $a + $b;
print "After assignment value of \$c = \$c\n";

$c += $a;
print "Value of \$c = \$c after statement \$c += \$a\n";

$c -= $a;
print "Value of \$c = \$c after statement \$c -= \$a\n";

$c *= $a;
print "Value of \$c = \$c after statement \$c *= \$a\n";
```
$c /= \$a;
print "Value of \$c = $c after statement \$c /= \$a\n";

$c %= \$a;
print "Value of \$c = $c after statement \$c %= \$a\n";

c = 2;
a = 4;
print "Value of \$a = $a and value of \$c = $c\n";
c **= \$a;
print "Value of \$c = $c after statement \$c **= \$a\n";

When the above code is executed, it produces the following result:

Value of $a = 10 and value of $b = 20
After assignment value of $c = 30
Value of $c = 40 after statement $c += $a
Value of $c = 30 after statement $c -= $a
Value of $c = 300 after statement $c *= $a
Value of $c = 30 after statement $c /= $a
Value of $c = 0 after statement $c %= $a
Value of $a = 4 and value of $c = 2
Value of $c = 16 after statement $c **= $a

Perl Bitwise Operators

Bitwise operator works on bits and perform bit by bit operation. Assume if $a = 60; and $b = 13; Now in binary format they will be as follows:

$a = 0011 1100
$b = 0000 1101

--------

$a&$b = 0000 1100
$a|$b = 0011 1101
$a^$b = 0011 0001
There are following Bitwise operators supported by Perl language, assume if $a = 60; and $b = 13:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Binary AND Operator copies a bit to the result if it exists in both operands.</td>
<td>($a &amp; $b) will give 12 which is 0000 1100</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```perl
$b = 13;

print "Value of $a = $a and value of $b = $b\n";

$c = $a & $b;
print "Value of $a & $b = $c\n";

$c = $a | $b;
print "Value of $a | $b = $c\n";

$c = $a ^ $b;
print "Value of $a ^ $b = $c\n";

$c = ~$a;
print "Value of ~$a = $c\n";

$c = $a << 2;
print "Value of $a << 2 = $c\n";

$c = $a >> 2;
print "Value of $a >> 2 = $c\n";
```

When the above code is executed, it produces the following result:

```
Value of $a = 60 and value of $b = 13
Value of $a & $b = 12
Value of $a | $b = 61
Value of $a ^ $b = 49
Value of ~$a = 18446744073709551555
Value of $a << 2 = 240
Value of $a >> 2 = 15
```
Perl Logical Operators

There are following logical operators supported by Perl language. Assume variable $a holds true and variable $b holds false then:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>Called Logical AND operator. If both the operands are true then then condition becomes true.</td>
<td>($a and $b) is false.</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>C-style Logical AND operator copies a bit to the result if it exists in both operands.</td>
<td>($a &amp;&amp; $b) is false.</td>
</tr>
<tr>
<td>or</td>
<td>Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.</td>
<td>($a or $b) is true.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.</td>
<td>not($a and $b) is true.</td>
</tr>
</tbody>
</table>

Example

Try the following example to understand all the logical operators available in Perl. Copy and paste the following Perl program in test.pl file and execute this program.

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

$a = true;
$b = false;

print "Value of \$a = $a and value of \$b = $b\n";

$c = ($a and $b);
print "Value of \$a and \$b = $c\n";
```
Perl

$c = ($a && $b);
print "Value of \$a && \$b = $c\n";

$c = ($a or $b);
print "Value of \$a or \$b = $c\n";

$c = ($a || $b);
print "Value of \$a || \$b = $c\n";

$a = 0;
$c = not($a);
print "Value of not(\$a)= $c\n";

When the above code is executed, it produces the following result:

<table>
<thead>
<tr>
<th>Value of $a = true and value of $b = false</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of $a and $b = false</td>
</tr>
<tr>
<td>Value of $a &amp;&amp; $b = false</td>
</tr>
<tr>
<td>Value of $a or $b = true</td>
</tr>
<tr>
<td>Value of $a</td>
</tr>
<tr>
<td>Value of not($a) = 1</td>
</tr>
</tbody>
</table>

Quote-like Operators

There are following Quote-like operators supported by Perl language. In the following table, a {} represents any pair of delimiters you choose.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>q{}</td>
<td>Encloses a string with-in single quotes</td>
<td>q{abcd} gives 'abcd'</td>
</tr>
<tr>
<td>qq{}</td>
<td>Encloses a string with-in double quotes</td>
<td>qq{abcd} &quot;abcd&quot;</td>
</tr>
<tr>
<td>qx{}</td>
<td>Encloses a string with-in invert quotes</td>
<td>qx{abcd}</td>
</tr>
</tbody>
</table>
Example
Try the following example to understand all the quote-like operators available in Perl. Copy and paste the following Perl program in test.pl file and execute this program.

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

$a = 10;

$b = q{a = $a};
print "Value of q{a = \$a} = $b\n";

$b = qq{a = $a};
print "Value of qq{a = \$a} = $b\n";

# unix command execution
$t = qx{date};
print "Value of qx{date} = $t\n";
```

When the above code is executed, it produces the following result:

Value of q{a = $a} = a = $a
Value of qq{a = $a} = a = 10
Value of qx{date} = Thu Feb 14 08:13:17 MST 2013

Miscellaneous Operators
There are following miscellaneous operators supported by Perl language. Assume variable a holds 10 and variable b holds 20 then:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Binary operator dot (.) concatenates two strings.</td>
<td>If $a=&quot;abc&quot;, $b=&quot;def&quot; then $a.$b will give &quot;abcdef&quot;</td>
</tr>
<tr>
<td>Operator</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>x</td>
<td>The repetition operator x returns a string consisting of the left operand repeated the number of times specified by the right operand.</td>
<td>('-' x 3) will give ----.</td>
</tr>
<tr>
<td>..</td>
<td>The range operator .. returns a list of values counting (up by ones) from the left value to the right value</td>
<td>(2..5) will give (2, 3, 4, 5)</td>
</tr>
<tr>
<td>++</td>
<td>Auto Increment operator increases integer value by one</td>
<td>$a++$ will give 11</td>
</tr>
<tr>
<td>--</td>
<td>Auto Decrement operator decreases integer value by one</td>
<td>$a--$ will give 9</td>
</tr>
<tr>
<td>-&gt;</td>
<td>The arrow operator is mostly used in dereferencing a method or variable from an object or a class name</td>
<td>$obj-&gt;a$ is an example to access variable $a$ from object $obj$.</td>
</tr>
</tbody>
</table>

**Example**

Try the following example to understand all the miscellaneous operators available in Perl. Copy and paste the following Perl program in test.pl file and execute this program.

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

$a = "abc";
$b = "def";

print "Value of \$a = $a and value of \$b = $b\n";

$c = $a . $b;
print "Value of \$a . \$b = $c\n";

$c = "-" x 3;
print "Value of \"-\" x 3 = $c\n";
```
@c = (2..5);
print "Value of (2..5) = @c\n";

$a = 10;
$b = 15;
print "Value of \$a = $a and value of \$b = $b\n";

$a++;
$c = $a;
print "Value of \$a after \$a++ = $c\n";

$b--;
$c = $b;
print "Value of \$b after \$b-- = $c\n";

When the above code is executed, it produces the following result:

Value of $a = abc and value of $b = def
Value of $a . $b = abcdef
Value of "-" x 3 = ---
Value of (2..5) = 2 3 4 5
Value of $a = 10 and value of $b = 15
Value of $a after $a++ = 11
Value of $b after $b-- = 14

We will explain --> operator when we will discuss about Perl Object and Classes.

**Perl Operators Precedence**

The following table lists all operators from highest precedence to lowest.

<table>
<thead>
<tr>
<th>left terms and list operators (leftward)</th>
<th>left -&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>nonassoc</td>
<td>++ --</td>
</tr>
<tr>
<td>right</td>
<td>**</td>
</tr>
<tr>
<td>right</td>
<td>! ~ \ and unary + and -</td>
</tr>
<tr>
<td>left</td>
<td>=~ !~</td>
</tr>
</tbody>
</table>
left * / % x
left + - .
left << >>

nonassoc  named unary operators
nonassoc  < > <= >= lt gt le ge
nonassoc  == != <=> eq ne cmp ~~
left &
left | ^
left &&
left || //

nonassoc  .. ...
right  ?:
right  += -= *= etc.
left , =>

nonassoc  list operators (rightward)
right  not
left and
left or xor

Example
Try the following example to understand all the perl operators precedence in Perl. Copy and paste the following Perl program in test.pl file and execute this program.

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

$a = 20;
$b = 10;
$c = 15;
$d = 5;
$e;

print "Value of \$a = $a, \$b = $b, \$c = $c and \$d = $d\n";

$e = ($a + $b) * $c / $d;
```
print "Value of (\$a + \$b) * \$c / \$d is = $e\n";

\$e = ((\$a + \$b) * \$c )/ \$d;
print "Value of ((\$a + \$b) * \$c) / \$d is = $e\n";

\$e = (\$a + \$b) * (\$c / \$d);
print "Value of (\$a + \$b) * (\$c / \$d ) is = $e\n";

\$e = \$a + (\$b * \$c )/ \$d;
print "Value of \$a + (\$b * \$c )/ \$d is = $e\n";

When the above code is executed, it produces the following result:

Value of \$a = 20, \$b = 10, \$c = 15 and \$d = 5
Value of (\$a + \$b) * \$c / \$d is = 90
Value of ((\$a + \$b) * \$c) / \$d is = 90
Value of (\$a + \$b) * (\$c / \$d ) is = 90
Value of \$a + (\$b * \$c )/ \$d is = 50
This chapter will give you the basic understanding on how to process and manipulate dates and times in Perl.

## Current Date and Time

Let's start with `localtime()` function, which returns values for the current date and time if given no arguments. Following is the 9-element list returned by the `localtime` function while using in list context:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec</td>
<td>seconds of minutes from 0 to 61</td>
</tr>
<tr>
<td>min</td>
<td>minutes of hour from 0 to 59</td>
</tr>
<tr>
<td>hour</td>
<td>hours of day from 0 to 24</td>
</tr>
<tr>
<td>mday</td>
<td>day of month from 1 to 31</td>
</tr>
<tr>
<td>mon</td>
<td>month of year from 0 to 11</td>
</tr>
<tr>
<td>year</td>
<td>year since 1900</td>
</tr>
<tr>
<td>wday</td>
<td>days since Sunday</td>
</tr>
<tr>
<td>yday</td>
<td>days since January 1st</td>
</tr>
<tr>
<td>isdst</td>
<td>hours of daylight savings time</td>
</tr>
</tbody>
</table>

Try the following example to print different elements returned by `localtime()` function:

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

@months = qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);
@days = qw(Sun Mon Tue Wed Thu Fri Sat Sun);

($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime();
print "$mday $months[$mon] $days[$wday]\n";
```

When the above code is executed, it produces the following result:

16 Feb Sat
If you will use localtime() function in scalar context, then it will return date and time from the current time zone set in the system. Try the following example to print current date and time in full format:

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

$datestring = localtime();
print "Local date and time $datestring\n";
```

When the above code is executed, it produces the following result:

```
Local date and time Sat Feb 16 06:50:45 2013
```

### GMT Time

The function gmtime() works just like localtime() function but the returned values are localized for the standard Greenwich time zone. When called in list context, $isdst, the last value returned by gmtime, is always 0. There is no Daylight Saving Time in GMT.

You should make a note on the fact that localtime() will return the current local time on the machine that runs the script and gmtime() will return the universal Greenwich Mean Time, or GMT (or UTC).

Try the following example to print the current date and time but on GMT scale:

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

$datestring = gmtime();
print "GMT date and time $datestring\n";
```

When the above code is executed, it produces the following result:

```
GMT date and time Sat Feb 16 13:50:45 2013
```

### Format Date & Time

You can use localtime() function to get a list of 9-elements and later you can use the printf() function to format date and time based on your requirements as follows:

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

```
Perl

```perl
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime();
printf("Time Format - HH:MM:SS\n");
printf("%02d:%02d:%02d", $hour, $min, $sec);
```

When the above code is executed, it produces the following result:

```
Time Format - HH:MM:SS
06:58:52
```

**Epoch time**

You can use the `time()` function to get epoch time, i.e., the numbers of seconds that have elapsed since a given date, in Unix is January 1, 1970.

```perl
#!/usr/local/bin/perl
$epoc = time();
print "Number of seconds since Jan 1, 1970 - $epoc\n";
```

When the above code is executed, it produces the following result:

```
Number of seconds since Jan 1, 1970 - 1361022130
```

You can convert a given number of seconds into date and time string as follows:

```perl
#!/usr/local/bin/perl
$datestring = localtime();
print "Current date and time $datestring\n";

$epoc = time();
$epoc = $epoc - 12 * 60 * 60;  # one day before of current date.

$datestring = localtime($epoc);
print "Yesterday's date and time $datestring\n";
```

When the above code is executed, it produces the following result:
Current date and time Sat Feb 16 07:05:39 2013
Yesterday's date and time Fri Feb 15 19:05:39 2013

### POSIX Function \texttt{strftime()}

You can use the POSIX function \texttt{strftime()} to format date and time with the help of the following table. Please note that the specifiers marked with an asterisk (*) are locale-dependent.

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Replaced by</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Abbreviated weekday name *</td>
<td>Thu</td>
</tr>
<tr>
<td>%A</td>
<td>Full weekday name *</td>
<td>Thursday</td>
</tr>
<tr>
<td>%b</td>
<td>Abbreviated month name *</td>
<td>Aug</td>
</tr>
<tr>
<td>%B</td>
<td>Full month name *</td>
<td>August</td>
</tr>
<tr>
<td>%c</td>
<td>Date and time representation *</td>
<td>Thu Aug 23 14:55:02 2001</td>
</tr>
<tr>
<td>%C</td>
<td>Year divided by 100 and truncated to integer (00-99)</td>
<td>20</td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month, zero-padded (01-31)</td>
<td>23</td>
</tr>
<tr>
<td>%D</td>
<td>Short MM/DD/YY date, equivalent to %m/%d/%y</td>
<td>08/23/01</td>
</tr>
<tr>
<td>%e</td>
<td>Day of the month, space-padded (1-31)</td>
<td>23</td>
</tr>
<tr>
<td>%F</td>
<td>Short YYYY-MM-DD date, equivalent to %Y-%m-%d</td>
<td>2001-08-23</td>
</tr>
<tr>
<td>%g</td>
<td>Week-based year, last two digits (00-99)</td>
<td>01</td>
</tr>
<tr>
<td>%G</td>
<td>Week-based year</td>
<td>2001</td>
</tr>
<tr>
<td>%h</td>
<td>Abbreviated month name * (same as %b)</td>
<td>Aug</td>
</tr>
<tr>
<td>Format</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>%H</td>
<td>Hour in 24h format (00-23)</td>
<td>14</td>
</tr>
<tr>
<td>%I</td>
<td>Hour in 12h format (01-12)</td>
<td>02</td>
</tr>
<tr>
<td>%j</td>
<td>Day of the year (001-366)</td>
<td>235</td>
</tr>
<tr>
<td>%m</td>
<td>Month as a decimal number (01-12)</td>
<td>08</td>
</tr>
<tr>
<td>%M</td>
<td>Minute (00-59)</td>
<td>55</td>
</tr>
<tr>
<td>%n</td>
<td>New-line character ('\n')</td>
<td></td>
</tr>
<tr>
<td>%p</td>
<td>AM or PM designation</td>
<td>PM</td>
</tr>
<tr>
<td>%r</td>
<td>12-hour clock time *</td>
<td>02:55:02 pm</td>
</tr>
<tr>
<td>%R</td>
<td>24-hour HH:MM time, equivalent to %H:%M</td>
<td>14:55</td>
</tr>
<tr>
<td>%S</td>
<td>Second (00-61)</td>
<td>02</td>
</tr>
<tr>
<td>%t</td>
<td>Horizontal-tab character ('\t')</td>
<td></td>
</tr>
<tr>
<td>%T</td>
<td>ISO 8601 time format (HH:MM:SS), equivalent to %H:%M:%S</td>
<td>14:55</td>
</tr>
<tr>
<td>%u</td>
<td>ISO 8601 weekday as number with Monday as 1 (1-7)</td>
<td>4</td>
</tr>
<tr>
<td>%U</td>
<td>Week number with the first Sunday as the first day of week one (00-53)</td>
<td>33</td>
</tr>
<tr>
<td>%V</td>
<td>ISO 8601 week number (00-53)</td>
<td>34</td>
</tr>
<tr>
<td>%w</td>
<td>Weekday as a decimal number with Sunday as 0 (0-6)</td>
<td>4</td>
</tr>
<tr>
<td>%W</td>
<td>Week number with the first Monday as the</td>
<td>34</td>
</tr>
</tbody>
</table>
Let's check the following example to understand the usage:

```perl
#!/usr/local/bin/perl
use POSIX qw strftime;

$datestring = strftime "%a %b %e %H:%M:%S %Y", localtime;
printf("date and time - $datestring\n");

# or for GMT formatted appropriately for your locale:
$datestring = strftime "%a %b %e %H:%M:%S %Y", gmtime;
printf("date and time - $datestring\n");
```

When the above code is executed, it produces the following result:

date and time - Sat Feb 16 07:10:23 2013

date and time - Sat Feb 16 14:10:23 2013
A Perl subroutine or function is a group of statements that together performs a task. You can divide up your code into separate subroutines. How you divide up your code among different subroutines is up to you, but logically the division usually is so each function performs a specific task.

Perl uses the terms subroutine, method and function interchangeably.

**Define and Call a Subroutine**

The general form of a subroutine definition in Perl programming language is as follows:

```perl
sub subroutine_name{
    body of the subroutine
}
```

The typical way of calling that Perl subroutine is as follows:

```perl
subroutine_name( list of arguments );
```

In versions of Perl before 5.0, the syntax for calling subroutines was slightly different as shown below. This still works in the newest versions of Perl, but it is not recommended since it bypasses the subroutine prototypes.

```perl
&subroutine_name( list of arguments );
```

Let's have a look into the following example, which defines a simple function and then call it. Because Perl compiles your program before executing it, it doesn't matter where you declare your subroutine.

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

# Function definition
sub Hello{
    print "Hello, World!\n";
}

# Function call
```
Hello();

When above program is executed, it produces the following result:

Hello, World!

**Passing Arguments to a Subroutine**

You can pass various arguments to a subroutine like you do in any other programming language and they can be accessed inside the function using the special array @_. Thus the first argument to the function is in $_[0], the second is in $_[1], and so on.

You can pass arrays and hashes as arguments like any scalar but passing more than one array or hash normally causes them to lose their separate identities. So we will use references (explained in the next chapter) to pass any array or hash.

Let's try the following example, which takes a list of numbers and then prints their average:

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

# Function definition
sub Average{
    # get total number of arguments passed.
    $n = scalar(@_);
    $sum = 0;

    foreach $item (@_){
        $sum += $item;
    }

    $average = $sum / $n;

    print "Average for the given numbers : $average\n";
}

# Function call
Average(10, 20, 30);
```
When above program is executed, it produces the following result:

Average for the given numbers: 20

**Passing Lists to Subroutines**

Because the `@_` variable is an array, it can be used to supply lists to a subroutine. However, because of the way in which Perl accepts and parses lists and arrays, it can be difficult to extract the individual elements from `@_`. If you have to pass a list along with other scalar arguments, then make list as the last argument as shown below:

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

# Function definition
sub PrintList{
    my @list = @_; 
    print "Given list is @list\n";
}

$a = 10;
@b = (1, 2, 3, 4);

# Function call with list parameter
PrintList($a, @b);
```

When above program is executed, it produces the following result:

Given list is 10 1 2 3 4

**Passing Hashes to Subroutines**

When you supply a hash to a subroutine or operator that accepts a list, then hash is automatically translated into a list of key/value pairs. For example:

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

# Function definition
sub PrintHash{
    my (%hash) = @_; 
}
```

```perl
    foreach my $key ( keys %hash ){
        my $value = $hash{$key};
        print "$key : $value\n";
    }

    %hash = ('name' => 'Tom', 'age' => 19);

    # Function call with hash parameter
    PrintHash(%hash);
```

When above program is executed, it produces the following result:

```
name : Tom
age : 19
```

### Returning Value from a Subroutine

You can return a value from subroutine like you do in any other programming language. If you are not returning a value from a subroutine then whatever calculation is last performed in a subroutine is automatically also the return value.

You can return arrays and hashes from the subroutine like any scalar but returning more than one array or hash normally causes them to lose their separate identities. So we will use references (explained in the next chapter) to return any array or hash from a function.

Let’s try the following example, which takes a list of numbers and then returns their average:

```
#!/usr/bin/perl

# Function definition
sub Average{
    # get total number of arguments passed.
    $n = scalar(@_);
    $sum = 0;
```
foreach $item (@_){
    $sum += $item;
}

$average = $sum / $n;

return $average;

# Function call
$num = Average(10, 20, 30);
print "Average for the given numbers : $num\n";
# Global variable
$string = "Hello, World!";

# Function definition
sub PrintHello{
    # Private variable for PrintHello function
    my $string;
    $string = "Hello, Perl!";
    print "Inside the function $string\n";
}
# Function call
PrintHello();
print "Outside the function $string\n";

When above program is executed, it produces the following result:

 внутри функции Hello, Perl!
вне функции Hello, World!

Темпорарные значения с помощью local()

Локальная переменная в основном используется, когда текущее значение переменной должно быть видимым для вызываемых подфункций. Локальная переменная дает временные значения глобальным (то есть пакетным) переменным. Это называется динамическим скопингом. Лексический скопинг реализован с помощью my, который работает более похоже на auto объявлений в C.

Если более одного переменной или выражения передается в локальном, они должны быть расположены в скобках. Этот оператор работает, сохраняя текущие значения этих переменных на скрытом стеке и восстанавливая их при выходе из блока, функции, или eval.

Проверим следующий пример, чтобы различить глобальные и локальные переменные:

#!/usr/bin/perl

# Global variable
$string = "Hello, World!";
sub PrintHello{
    # Private variable for PrintHello function
    local $string;
    $string = "Hello, Perl!";
    PrintMe();
    print "Inside the function PrintHello $string\n";
}
sub PrintMe{
    print "Inside the function PrintMe $string\n";
}

# Function call
PrintHello();
print "Outside the function $string\n";

When above program is executed, it produces the following result:

Inside the function PrintMe Hello, Perl!
Inside the function PrintHello Hello, Perl!
Outside the function Hello, World!

State Variables via state()

There are another type of lexical variables, which are similar to private variables but they maintain their state and they do not get reinitialized upon multiple calls of the subroutines. These variables are defined using the state operator and available starting from Perl 5.9.4.

Let's check the following example to demonstrate the use of state variables:

#!/usr/bin/perl

use feature 'state';

sub PrintCount{
    state $count = 0; # initial value
When above program is executed, it produces the following result:

Value of counter is 0
Value of counter is 1
Value of counter is 2
Value of counter is 3
Value of counter is 4

Prior to Perl 5.10, you would have to write it like this:

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

{  
  my $count = 0;  # initial value

  sub PrintCount {  
    print "Value of counter is $count\n";
    $count++;
  }
}

for (1..5){
  PrintCount();
}
```

**Subroutine Call Context**

The context of a subroutine or statement is defined as the type of return value that is expected. This allows you to use a single function that returns different
values based on what the user is expecting to receive. For example, the following localtime() returns a string when it is called in scalar context, but it returns a list when it is called in list context.

```perl
my $datestring = localtime( time );
```

In this example, the value of $timestr is now a string made up of the current date and time, for example, Thu Nov 30 15:21:33 2000. Conversely:

```perl
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime(time);
```

Now the individual variables contain the corresponding values returned by localtime() subroutine.
A Perl reference is a scalar data type that holds the location of another value which could be scalar, arrays, or hashes. Because of its scalar nature, a reference can be used anywhere, a scalar can be used.

You can construct lists containing references to other lists, which can contain references to hashes, and so on. This is how the nested data structures are built in Perl.

### Create References

It is easy to create a reference for any variable, subroutine or value by prefixing it with a backslash as follows:

```perl
$scalarref = \$foo;
$arrayref = @ARGV;
$hashref = %ENV;
$coderef = \&handler;
$globref = *foo;
```

You cannot create a reference on an I/O handle (filehandle or dirhandle) using the backslash operator but a reference to an anonymous array can be created using the square brackets as follows:

```perl
$arrayref = [1, 2, ['a', 'b', 'c']];
```

Similar way you can create a reference to an anonymous hash using the curly brackets as follows:

```perl
$hashref = {
    'Adam' => 'Eve',
    'Clyde' => 'Bonnie',
};
```

A reference to an anonymous subroutine can be created by using sub without a subname as follows:

```perl
$coderef = sub { print "Boink!\n"; }
```
Dereferencing

Dereferencing returns the value from a reference point to the location. To dereference a reference simply use $, @ or % as prefix of the reference variable depending on whether the reference is pointing to a scalar, array, or hash. Following is the example to explain the concept:

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

$var = 10;

# Now $r has reference to $var scalar.
$r = \$var;

# Print value available at the location stored in $r.
print "Value of $var is : ", $r, "\n";

@var = (1, 2, 3);
# Now $r has reference to @var array.
$r = @var;
# Print values available at the location stored in $r.
print "Value of @var is : ", @r, "\n";

%var = ("key1" => 10, "key2" => 20);
# Now $r has reference to %var hash.
$r = \%var;
# Print values available at the location stored in $r.
print "Value of %var is : ", %r, "\n";

When above program is executed, it produces the following result:

Value of 10 is : 10
Value of 1 2 3 is : 123
Value of %var is : key220key110

If you are not sure about a variable type, then its easy to know its type using ref, which returns one of the following strings if its argument is a reference. Otherwise, it returns false:
Let's try the following example:

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

$var = 10;
$r = $var;
print "Reference type in r : ", ref($r), "\n";

@var = (1, 2, 3);
$r = @var;
print "Reference type in r : ", ref($r), "\n";

%var = ('key1' => 10, 'key2' => 20);
$r = %var;
print "Reference type in r : ", ref($r), "\n";
```

When above program is executed, it produces the following result:

Reference type in r : SCALAR
Reference type in r : ARRAY
Reference type in r : HASH

**Circular References**

A circular reference occurs when two references contain a reference to each other. You have to be careful while creating references otherwise a circular reference can lead to memory leaks. Following is an example:

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

my $foo = 100;
```
$foo = \$foo;

print "Value of foo is : ", $$foo, "\n";

When above program is executed, it produces the following result:

Value of foo is : REF(0x9aae38)

References to Functions

This might happen if you need to create a signal handler so you can produce a reference to a function by preceding that function name with \& and to dereference that reference you simply need to prefix reference variable using ampersand &. Following is an example:

#!/usr/bin/perl

# Function definition
sub PrintHash{
  my (%hash) = @_;

  foreach $item (%hash){
    print "Item : $item\n";
  }
}
%hash = ('name' => 'Tom', 'age' => 19);

# Create a reference to above function.
$cref = \&PrintHash;

# Function call using reference.
&$cref(%hash);

When above program is executed, it produces the following result:

Item : name
Item : Tom
Item : age
Item : 19
Perl uses a writing template called a ‘format’ to output reports. To use the format feature of Perl, you have to define a format first and then you can use that format to write formatted data.

**Define a Format**

Following is the syntax to define a Perl format:

```perl
format FormatName =
  fieldline
  value_one, value_two, value_three
  fieldline
  value_one, value_two
 .
```

Here `FormatName` represents the name of the format. The `fieldline` is the specific way, the data should be formatted. The values lines represent the values that will be entered into the field line. You end the format with a single period.

Next `fieldline` can contain any text or fieldholders. The fieldholders hold space for data that will be placed there at a later date. A fieldholder has the format:

```
@<<<<
```

This fieldholder is left-justified, with a field space of 5. You must count the `@` sign and the `<` signs to know the number of spaces in the field. Other field holders include:

```
@>>>> right-justified
@|||| centered
@####.## numeric field holder
@* multiline field holder
```

An example format would be:

```perl
format EMPLOYEE =
  =*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*
In this example, $name would be written as left justify within 22 character spaces and after that age will be written in two spaces.

**Using the Format**

In order to invoke this format declaration, we would use the `write` keyword:

```perl
write EMPLOYEE;
```

The problem is that the format name is usually the name of an open file handle, and the write statement will send the output to this file handle. As we want the data sent to the STDOUT, we must associate EMPLOYEE with the STDOUT filehandle. First, however, we must make sure that that STDOUT is our selected file handle, using the `select()` function.

```perl
select(STDOUT);
```

We would then associate EMPLOYEE with STDOUT by setting the new format name with STDOUT, using the special variable `~` or `$FORMAT_NAME` as follows:

```perl
~ = "EMPLOYEE"
```

When we now do a `write()`, the data would be sent to STDOUT. Remember: if you are going to write your report in any other file handle instead of STDOUT then you can use `select()` function to select that file handle and rest of the logic will remain the same.

Let's take the following example. Here we have hard coded values just for showing the usage. In actual usage you will read values from a file or database to generate actual reports and you may need to write final report again into a file.

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

format EMPLOYEE =

===================================
@<<<<<<<<<<<<<<<<<<<<<<
@<<
```

```perl
$name $age
@####@##
$salary
===================================
.
```
$name $age
@#####.##
$salary
===================================.

select(STDOUT);
$~ = EMPLOYEE;

@n = ("Ali", "Raza", "Jaffer");
@a = (20, 30, 40);
@s = (2000.00, 2500.00, 4000.00);

$i = 0;
foreach (@n){
    $name = $_;
    $age = $a[$i];
    $salary = $s[$i++];
    write;
}

When executed, this will produce the following result:

===================================
Ali 20
2000.00
===================================
===================================
Raza 30
2500.00
===================================
===================================
Jaffer 40
4000.00
Define a Report Header

Everything looks fine. But you would be interested in adding a header to your report. This header will be printed on top of each page. It is very simple to do this. Apart from defining a template you would have to define a header and assign it to $^ or $FORMAT_TOP_NAME variable:

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

format EMPLOYEE =

===================================
@<<<<<<<<<<<<@<
$name $age
@#####.##
$salary

===================================.

format EMPLOYEE_TOP =

===================================
Name Age

===================================.

select(STDOUT);
$~ = EMPLOYEE;
$^ = EMPLOYEE_TOP;

@n = ("Ali", "Raza", "Jaffer");
@a = (20, 30, 40);
@s = (2000.00, 2500.00, 4000.00);

$i = 0;
```
foreach (@n) {
    $name = $_;
    $age = $a[$i];
    $salary = $s[$i++];
    write;
}

Now your report will look like:

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ali</td>
<td>20</td>
<td>2000.00</td>
</tr>
<tr>
<td>Raza</td>
<td>30</td>
<td>2500.00</td>
</tr>
<tr>
<td>Jaffer</td>
<td>40</td>
<td>4000.00</td>
</tr>
</tbody>
</table>

Define a Pagination

What about if your report is taking more than one page? You have a solution for that, simply use $% or $FORMAT_PAGE_NUMBER vairable along with header as follows:

format EMPLOYEE_TOP =
                      =
| Name | Age | Page | @< $% 
|------|-----|------|------
| Ali  | 20  | 0    |      |
| Raza | 30  | 1    |      |
| Jaffer | 40  | 2    |      |
## Number of Lines on a Page

You can set the number of lines per page using special variable $= ( or $FORMAT_LINES_PER_PAGE). By default $= will be 60.

## Define a Report Footer

While $^ or $FORMAT_TOP_NAME contains the name of the current header format, there is no corresponding mechanism to automatically do the same thing for a footer. If you have a fixed-size footer, you can get footers by checking variable $- or $FORMAT_LINES_LEFT before each write() and print the footer yourself if necessary using another format defined as follows:

```perl
format EMPLOYEE_BOTTOM = End of Page @< $% .
```

---

### Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Page 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ali</td>
<td>20</td>
<td>2000.00</td>
</tr>
<tr>
<td>Raza</td>
<td>30</td>
<td>2500.00</td>
</tr>
<tr>
<td>Jaffer</td>
<td>40</td>
<td>4000.00</td>
</tr>
</tbody>
</table>

---

### Perl

Now your output will look like as follows:

135
For a complete set of variables related to formatting, please refer to the Perl Special Variables section.
The basics of handling files are simple: you associate a **filehandle** with an external entity (usually a file) and then use a variety of operators and functions within Perl to read and update the data stored within the data stream associated with the filehandle.

A filehandle is a named internal Perl structure that associates a physical file with a name. All filehandles are capable of read/write access, so you can read from and update any file or device associated with a filehandle. However, when you associate a filehandle, you can specify the mode in which the filehandle is opened.

Three basic file handles are - **STDIN**, **STDOUT**, and **STDERR**, which represent standard input, standard output and standard error devices respectively.

### Opening and Closing Files

There are following two functions with multiple forms, which can be used to open any new or existing file in Perl.

```perl
open FILEHANDLE, EXPR
open FILEHANDLE

sysopen FILEHANDLE, FILENAME, MODE, PERMS
sysopen FILEHANDLE, FILENAME, MODE
```

Here FILEHANDLE is the file handle returned by the `open` function and EXPR is the expression having file name and mode of opening the file.

### Open Function

Following is the syntax to open **file.txt** in read-only mode. Here less than `<` sign indicates that file has to be opened in read-only mode.

```perl
open(DATA, "<file.txt");
```

Here DATA is the file handle, which will be used to read the file. Here is the example, which will open a file and will print its content over the screen.

```perl
#!/usr/bin/perl
```
Following is the syntax to open file.txt in writing mode. Here less than > sign indicates that file has to be open in the writing mode.

```
open(DATA, ">file.txt") or die "Couldn't open file file.txt, $!";
```

This example actually truncates (empties) the file before opening it for writing, which may not be the desired effect. If you want to open a file for reading and writing, you can put a plus sign before the > or < characters.

For example, to open a file for updating without truncating it:

```
open(DATA, "+<file.txt") or die "Couldn't open file file.txt, $!";
```

To truncate the file first:

```
open DATA, "+>file.txt" or die "Couldn't open file file.txt, $!";
```

You can open a file in the append mode. In this mode, writing point will be set to the end of the file.

```
open(DATA,">>file.txt") || die "Couldn't open file file.txt, $!";
```

A double >> opens the file for appending, placing the file pointer at the end, so that you can immediately start appending information. However, you can't read from it unless you also place a plus sign in front of it:

```
open(DATA,"+>>file.txt") || die "Couldn't open file file.txt, $!";
```

Following is the table, which gives the possible values of different modes

<table>
<thead>
<tr>
<th>Entities</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; or r</td>
<td>Read Only Access</td>
</tr>
<tr>
<td>&gt; or w</td>
<td>Creates, Writes, and Truncates</td>
</tr>
<tr>
<td>&gt;&gt; or a</td>
<td>Writes, Appends, and Creates</td>
</tr>
</tbody>
</table>
 Sysopen Function

The **sysopen** function is similar to the main open function, except that it uses the system **open()** function, using the parameters supplied to it as the parameters for the system function:

For example, to open a file for updating, emulating the **+<filename** format from open:

```perl
sysopen(DATA, "file.txt", O_RDWR);
```

Or to truncate the file before updating:

```perl
sysopen(DATA, "file.txt", O_RDWR|O_TRUNC);
```

You can use **O_CREAT** to create a new file and **O_WRONLY** to open file in write only mode and **O_RDONLY** - to open file in read only mode.

The **PERMS** argument specifies the file permissions for the file specified, if it has to be created. By default it takes **0x666**.

Following is the table, which gives the possible values of MODE.

<table>
<thead>
<tr>
<th>Entities</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>O_RDWR</td>
<td>Read and Write</td>
</tr>
<tr>
<td>O_RDONLY</td>
<td>Read Only</td>
</tr>
<tr>
<td>O_WRONLY</td>
<td>Write Only</td>
</tr>
<tr>
<td>O_CREAT</td>
<td>Create the file</td>
</tr>
<tr>
<td>O_APPEND</td>
<td>Append the file</td>
</tr>
<tr>
<td>O_TRUNC</td>
<td>Truncate the file</td>
</tr>
</tbody>
</table>
Close Function
To close a filehandle, and therefore disassociate the filehandle from the corresponding file, you use the `close` function. This flushes the filehandle's buffers and closes the system's file descriptor.

```
close FILEHANDLE
```

If no FILEHANDLE is specified, then it closes the currently selected filehandle. It returns true only if it could successfully flush the buffers and close the file.

```
close(DATA) || die "Couldn't close file properly";
```

Reading and Writing Files
Once you have an open filehandle, you need to be able to read and write information. There are a number of different ways of reading and writing data into the file.

The `<FILEHANDLE>` Operator
The main method of reading the information from an open filehandle is the `<FILEHANDLE>` operator. In a scalar context, it returns a single line from the filehandle. For example:

```
#!/usr/bin/perl

print "What is your name?\n";
$name = <STDIN>;
print "Hello $name\n";
```

When you use the `<FILEHANDLE>` operator in a list context, it returns a list of lines from the specified filehandle. For example, to import all the lines from a file into an array:

```
#!/usr/bin/perl
```
open(DATA,"<import.txt") or die "Can't open data";
@lines = <DATA>;
close(DATA);

**getc Function**

The getc function returns a single character from the specified FILEHANDLE, or STDOUT if none is specified:

```
getc FILEHANDLE

getc
```

If there was an error, or the filehandle is at end of file, then undef is returned instead.

**read Function**

The read function reads a block of information from the buffered filehandle: This function is used to read binary data from the file.

```
read FILEHANDLE, SCALAR, LENGTH, OFFSET
read FILEHANDLE, SCALAR, LENGTH
```

The length of the data read is defined by LENGTH, and the data is placed at the start of SCALAR if no OFFSET is specified. Otherwise data is placed after OFFSET bytes in SCALAR. The function returns the number of bytes read on success, zero at end of file, or undef if there was an error.

**print Function**

For all the different methods used for reading information from filehandles, the main function for writing information back is the print function.

```
print FILEHANDLE LIST
print LIST
print
```

The print function prints the evaluated value of LIST to FILEHANDLE, or to the current output filehandle (STDOUT by default). For example:

```
print "Hello World!\n";
```
Copying Files

Here is the example, which opens an existing file file1.txt and read it line by line and generate another copy file file2.txt.

```perl
#!/usr/bin/perl
# Open file to read
open(DATA1, "<file1.txt");

# Open new file to write
open(DATA2, ">file2.txt");

# Copy data from one file to another.
while(<DATA1>)
{
   print DATA2 $_;
}

close( DATA1 );

close( DATA2 );
```

Renaming a file

Here is an example, which shows how we can rename a file file1.txt to file2.txt. Assuming file is available in /usr/test directory.

```perl
#!/usr/bin/perl
rename ("/usr/test/file1.txt", "/usr/test/file2.txt");
```

This function `rename` takes two arguments and it just renames the existing file.

Deleting an Existing File

Here is an example, which shows how to delete a file file1.txt using the `unlink` function.

```perl
#!/usr/bin/perl
```
unlink("/usr/test/file1.txt");

Positioning inside a File

You can use the `tell` function to know the current position of a file and the `seek` function to point a particular position inside the file.

**tell Function**

The first requirement is to find your position within a file, which you do using the `tell` function:

```
tell FILEHANDLE
```

This returns the position of the file pointer, in bytes, within `FILEHANDLE` if specified, or the current default selected filehandle if none is specified.

**seek Function**

The `seek` function positions the file pointer to the specified number of bytes within a file:

```
seek FILEHANDLE, POSITION, WHENCE
```

The function uses the fseek system function, and you have the same ability to position relative to three different points: the start, the end, and the current position. You do this by specifying a value for `WHENCE`.

Zero sets the positioning relative to the start of the file. For example, the line sets the file pointer to the 256th byte in the file.

```
seek DATA, 256, 0;
```

**File Information**

You can test certain features very quickly within Perl using a series of test operators known collectively as `-X` tests. For example, to perform a quick test of the various permissions on a file, you might use a script like this:

```
#!/usr/bin/perl

my $file = "/usr/test/file1.txt";
my (@description, $size);
```
if (-e $file)
{
    push @description, 'binary' if (-B _);
    push @description, 'a socket' if (-S _);
    push @description, 'a text file' if (-T _);
    push @description, 'a block special file' if (-b _);
    push @description, 'a character special file' if (-c _);
    push @description, 'a directory' if (-d _);
    push @description, 'executable' if (-x _);
    push @description, (($size = -s _) ? "$size bytes" : 'empty');
    print "$file is ", join(',', ',@description),"\n";
}

Here is the list of features, which you can check for a file or directory:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>-A</td>
<td>Script start time minus file last access time, in days.</td>
</tr>
<tr>
<td>-B</td>
<td>Is it a binary file?</td>
</tr>
<tr>
<td>-C</td>
<td>Script start time minus file last inode change time, in days.</td>
</tr>
<tr>
<td>-M</td>
<td>Script start time minus file modification time, in days.</td>
</tr>
<tr>
<td>-O</td>
<td>Is the file owned by the real user ID?</td>
</tr>
<tr>
<td>-R</td>
<td>Is the file readable by the real user ID or real group?</td>
</tr>
<tr>
<td>-S</td>
<td>Is the file a socket?</td>
</tr>
<tr>
<td>-T</td>
<td>Is it a text file?</td>
</tr>
<tr>
<td>-W</td>
<td>Is the file writable by the real user ID or real group?</td>
</tr>
<tr>
<td>-X</td>
<td>Is the file executable by the real user ID or real group?</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>-b</td>
<td>Is it a block special file?</td>
</tr>
<tr>
<td>-c</td>
<td>Is it a character special file?</td>
</tr>
<tr>
<td>-d</td>
<td>Is the file a directory?</td>
</tr>
<tr>
<td>-e</td>
<td>Does the file exist?</td>
</tr>
<tr>
<td>-f</td>
<td>Is it a plain file?</td>
</tr>
<tr>
<td>-g</td>
<td>Does the file have the setgid bit set?</td>
</tr>
<tr>
<td>-k</td>
<td>Does the file have the sticky bit set?</td>
</tr>
<tr>
<td>-l</td>
<td>Is the file a symbolic link?</td>
</tr>
<tr>
<td>-o</td>
<td>Is the file owned by the effective user ID?</td>
</tr>
<tr>
<td>-p</td>
<td>Is the file a named pipe?</td>
</tr>
<tr>
<td>-r</td>
<td>Is the file readable by the effective user or group ID?</td>
</tr>
<tr>
<td>-s</td>
<td>Returns the size of the file, zero size = empty file.</td>
</tr>
<tr>
<td>-t</td>
<td>Is the filehandle opened by a TTY (terminal)?</td>
</tr>
<tr>
<td>-u</td>
<td>Does the file have the setuid bit set?</td>
</tr>
<tr>
<td>-w</td>
<td>Is the file writable by the effective user or group ID?</td>
</tr>
<tr>
<td>-x</td>
<td>Is the file executable by the effective user or group ID?</td>
</tr>
<tr>
<td>-z</td>
<td>Is the file size zero?</td>
</tr>
</tbody>
</table>
Following are the standard functions used to play with directories.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>opendir DIRHANDLE, EXPR</td>
<td># To open a directory</td>
</tr>
<tr>
<td>readdir DIRHANDLE</td>
<td># To read a directory</td>
</tr>
<tr>
<td>rewinddir DIRHANDLE</td>
<td># Positioning pointer to the beginning</td>
</tr>
<tr>
<td>telldir DIRHANDLE</td>
<td># Returns current position of the dir</td>
</tr>
<tr>
<td>seekdir DIRHANDLE, POS</td>
<td># Pointing pointer to POS inside dir</td>
</tr>
<tr>
<td>closedir DIRHANDLE</td>
<td># Closing a directory</td>
</tr>
</tbody>
</table>

**Display all the Files**

There are various ways to list down all the files available in a particular directory. First let's use the simple way to get and list down all the files using the `glob` operator:

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

# Display all the files in /tmp directory.
$dir = "/tmp/*";
my @files = glob( $dir );

foreach (@files ){
    print $_ . "\n";
}

# Display all the C source files in /tmp directory.
$dir = "/tmp/*.c";
@files = glob( $dir );

foreach (@files ){
    print $_ . "\n";
}
```
# Display all the hidden files.
$dir = "/tmp/.*";
@files = glob( $dir );
foreach (@files ){
    print ".\n";
}

# Display all the files from /tmp and /home directories.
$dir = "/tmp/* /home/*/";
@files = glob( $dir );
foreach (@files ){
    print ".\n";
}

Here is another example, which opens a directory and list out all the files available inside this directory.

#!/usr/bin/perl
opendir (DIR, '.') or die "Couldn't open directory, $!";
while ($file = readdir DIR) {
    print "$file\n";
}
closedir DIR;

One more example to print the list of C source files you might use is:

#!/usr/bin/perl
opendir(DIR, '.') or die "Couldn't open directory, $!";
foreach (sort grep(/^.*\.c$/, readdir(DIR))){
    print ".\n";
}
closedir DIR;
Create new Directory

You can use `mkdir` function to create a new directory. You will need to have the required permission to create a directory.

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

$dir = "/tmp/perl";

# This creates perl directory in /tmp directory.
mkdir( $dir ) or die "Couldn't create $dir directory, $!";
print "Directory created successfully\n";
```

Remove a directory

You can use `rmdir` function to remove a directory. You will need to have the required permission to remove a directory. Additionally this directory should be empty before you try to remove it.

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

$dir = "/tmp/perl";

# This removes perl directory from /tmp directory.
rmdir( $dir ) or die "Couldn't remove $dir directory, $!";
print "Directory removed successfully\n";
```

Change a Directory

You can use `chdir` function to change a directory and go to a new location. You will need to have the required permission to change a directory and go inside the new directory.

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

$dir = "/home";

# This changes perl directory and moves you inside /home directory.
chdir( $dir ) or die "Couldn't go inside $dir directory, $!";
```
print "Your new location is $dir\n";
The execution and the errors always go together. If you are opening a file which does not exist, then if you did not handle this situation properly then your program is considered to be of bad quality.

The program stops if an error occurs. So a proper error handling is used to handle various type of errors, which may occur during a program execution and take appropriate action instead of halting program completely.

You can identify and trap an error in a number of different ways. Its very easy to trap errors in Perl and then handling them properly. Here are few methods which can be used.

**The if statement**

The **if statement** is the obvious choice when you need to check the return value from a statement; for example:

```perl
if(open(DATA, $file)){
    ...
}else{
    die "Error: Couldn't open the file - $!";
}
```

Here variable $! returns the actual error message. Alternatively, we can reduce the statement to one line in situations where it makes sense to do so; for example:

```perl
open(DATA, $file) || die "Error: Couldn't open the file $!";
```

**The unless Function**

The **unless** function is the logical opposite to if: statements can completely bypass the success status and only be executed if the expression returns false. For example:

```perl
unless(chdir("/etc"){
    die "Error: Can't change directory - $!";
}
```
The `unless` statement is best used when you want to raise an error or alternative only if the expression fails. The statement also makes sense when used in a single-line statement:

```perl
die "Error: Can't change directory!: $!" unless(chdir("/etc"));
```

Here we die only if the chdir operation fails, and it reads nicely.

### The ternary Operator

For very short tests, you can use the conditional operator `?:`:

```perl
print(exists($hash{value}) ? 'There' : 'Missing',"\n");
```

It's not quite so clear here what we are trying to achieve, but the effect is the same as using an `if` or `unless` statement. The conditional operator is best used when you want to quickly return one of the two values within an expression or statement.

### The warn Function

The warn function just raises a warning, a message is printed to STDERR, but no further action is taken. So it is more useful if you just want to print a warning for the user and proceed with rest of the operation:

```perl
chdir('/etc') or warn "Can't change directory";
```

### The die Function

The die function works just like warn, except that it also calls exit. Within a normal script, this function has the effect of immediately terminating execution. You should use this function in case it is useless to proceed if there is an error in the program:

```perl
chdir('/etc') or die "Can't change directory";
```

### Errors within Modules

There are two different situations we should be able to handle:

- Reporting an error in a module that quotes the module's filename and line number - this is useful when debugging a module, or when you specifically want to raise a module-related, rather than script-related, error.
• Reporting an error within a module that quotes the caller's information so that you can debug the line within the script that caused the error. Errors raised in this fashion are useful to the end-user, because they highlight the error in relation to the calling script's origination line.

The `warn` and `die` functions work slightly differently than you would expect when called from within a module. For example, the simple module:

```
package T;

require Exporter;
@ISA = qw/Exporter/;
@EXPORT = qw/function/;
use Carp;

sub function {
    warn "Error in module!";
}
1;
```

When called from a script like below:

```
use T;
function();
```

It will produce the following result:

```
Error in module! at T.pm line 9.
```

This is more or less what you might expected, but not necessarily what you want. From a module programmer's perspective, the information is useful because it helps to point to a bug within the module itself. For an end-user, the information provided is fairly useless, and for all but the hardened programmer, it is completely pointless.

The solution for such problems is the Carp module, which provides a simplified method for reporting errors within modules that return information about the calling script. The Carp module provides four functions: `carp`, `cluck`, `croak`, and `confess`. These functions are discussed below.
The carp Function

The carp function is the basic equivalent of warn and prints the message to STDERR without actually exiting the script and printing the script name.

```perl
package T;

require Exporter;
@ISA = qw/Exporter/;
@EXPORT = qw/function/;
use Carp;

sub function {
    carp "Error in module!";
}
1;
```

When called from a script like below:

```perl
use T;
function();
```

It will produce the following result:

```
Error in module! at test.pl line 4
```

The cluck Function

The cluck function is a sort of supercharged carp, it follows the same basic principle but also prints a stack trace of all the modules that led to the function being called, including the information on the original script.

```perl
package T;

require Exporter;
@ISA = qw/Exporter/;
@EXPORT = qw/function/;
use Carp qw(cluck);
```
sub function {
    cluck "Error in module!";
}
1;

When called from a script like below:

use T;
function();

It will produce the following result:

Error in module! at T.pm line 9
    T::function() called at test.pl line 4

**The croak Function**

The `croak` function is equivalent to `die`, except that it reports the caller one level up. Like `die`, this function also exits the script after reporting the error to STDERR:

package T;

require Exporter;
@ISA = qw/Exporter/;
@EXPORT = qw/function/;
use Carp;

sub function {
    croak "Error in module!";
}
1;

When called from a script like below:

use T;
function();

It will produce the following result:
Error in module! at test.pl line 4

As with carp, the same basic rules apply regarding the including of line and file information according to the warn and die functions.

The confess Function

The confess function is like cluck; it calls die and then prints a stack trace all the way up to the origination script.

```perl
package T;

require Exporter;
@ISA = qw/Exporter/;
@EXPORT = qw/function/;
use Carp;

sub function {
    confess "Error in module!";
}
1;
```

When called from a script like below:

```perl
use T;
function();
```

It will produce the following result:

```
Error in module! at T.pm line 9
    T::function() called at test.pl line 4
```
There are some variables which have a predefined and special meaning in Perl. They are the variables that use punctuation characters after the usual variable indicator ($, @, or %), such as $ (explained below).

Most of the special variables have an English like long name, e.g., Operating System Error variable $! can be written as $OS_ERROR. But if you are going to use English like names, then you would have to put one line `use English;` at the top of your program file. This guides the interpreter to pickup exact meaning of the variable.

The most commonly used special variable is $, which contains the default input and pattern-searching string. For example, in the following lines:

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

foreach ('hickory', 'dickory', 'doc') {
    print $;
    print "\n";
}
```

When executed, this will produce the following result:

```
hickory
dickory
doc
```

Again, let's check the same example without using $ variable explicitly:

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

foreach ('hickory', 'dickory', 'doc') {
    print;
    print "\n";
}
```

When executed, this will also produce the following result:

```
hickory
```
The first time the loop is executed, "hickory" is printed. The second time around, "dickory" is printed, and the third time, "doc" is printed. That's because in each iteration of the loop, the current string is placed in $_, and is used by default by print. Here are the places where Perl will assume $_ even if you don't specify it:

- Various unary functions, including functions like ord and int, as well as the all file tests (-f, -d) except for -t, which defaults to STDIN.
- Various list functions like print and unlink.
- The pattern-matching operations m//, s///, and tr/// when used without an =~ operator.
- The default iterator variable in a foreach loop if no other variable is supplied.
- The implicit iterator variable in the grep and map functions.
- The default place to put an input record when a line-input operation's result is tested by itself as the sole criterion of a while test (i.e., ). Note that outside of a while test, this will not happen.

Special Variable Types

Based on the usage and nature of special variables, we can categorize them in the following categories:

- Global Scalar Special Variables.
- Global Array Special Variables.
- Global Hash Special Variables.
- Global Special Filehandles.
- Global Special Constants.
- Regular Expression Special Variables.
- Filehandle Special Variables.

Global Scalar Special Variables

Here is the list of all the scalar special variables. We have listed corresponding english like names along with the symbolic names.

<p>| $_       | The default input and pattern- |</p>
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$ARG</code></td>
<td>searching space.</td>
</tr>
<tr>
<td><code>.</code></td>
<td>The current input line number of the last filehandle that was read. An explicit close on the filehandle resets the line number.</td>
</tr>
<tr>
<td><code>$NR</code></td>
<td>The input record separator; newline by default. If set to the null string, it treats blank lines as delimiters.</td>
</tr>
<tr>
<td><code>/</code></td>
<td>The output record separator for the print operator.</td>
</tr>
<tr>
<td><code>$RS</code></td>
<td>The output field separator for the print operator.</td>
</tr>
<tr>
<td><code>\</code></td>
<td>The output record separator for the print operator.</td>
</tr>
<tr>
<td><code>$ORS</code></td>
<td>Like <code>\</code>, except that it applies to list values interpolated into a double-quoted string (or similar interpreted string). Default is a space.</td>
</tr>
<tr>
<td><code>$LIST_SEPARATOR</code></td>
<td>The subscript separator for multidimensional array emulation. Default is <code>&quot;\034&quot;</code>.</td>
</tr>
<tr>
<td><code>;</code></td>
<td>The current value of the write</td>
</tr>
<tr>
<td><code>$FORMAT_FORMFEED</code></td>
<td>What a format outputs to perform a formfeed. Default is <code>&quot;\f&quot;</code>.</td>
</tr>
<tr>
<td><code>$FORMAT_LINE_BREAK_CHARACTERS</code></td>
<td>The current set of characters after which a string may be broken to fill continuation fields (starting with ^) in a format. Default is <code>&quot;\n&quot;</code>.</td>
</tr>
<tr>
<td><code>^A</code></td>
<td>The current value of the write</td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>$ACCUMULATOR</td>
<td>accumulator for format lines.</td>
</tr>
<tr>
<td>$#</td>
<td>Contains the output format for printed numbers (deprecated).</td>
</tr>
<tr>
<td>$OFMT</td>
<td></td>
</tr>
<tr>
<td>$?</td>
<td>The status returned by the last pipe close, backtick (``) command, or system operator.</td>
</tr>
<tr>
<td>$CHILD_ERROR</td>
<td></td>
</tr>
<tr>
<td>$!</td>
<td>If used in a numeric context, yields the current value of the errno variable, identifying the last system call error. If used in a string context, yields the corresponding system error string.</td>
</tr>
<tr>
<td>$OS_ERROR or $ERRNO</td>
<td></td>
</tr>
<tr>
<td>$@</td>
<td>The Perl syntax error message from the last eval command.</td>
</tr>
<tr>
<td>$EVAL_ERROR</td>
<td></td>
</tr>
<tr>
<td>$$</td>
<td>The pid of the Perl process running this script.</td>
</tr>
<tr>
<td>$PROCESS_ID or $PID</td>
<td></td>
</tr>
<tr>
<td>$&lt;</td>
<td>The real user ID (uid) of this process.</td>
</tr>
<tr>
<td>$REAL_USER_ID or $UID</td>
<td></td>
</tr>
<tr>
<td>$&gt;</td>
<td>The effective user ID of this process.</td>
</tr>
<tr>
<td>$EFFECTIVE_USER_ID or $EUID</td>
<td></td>
</tr>
<tr>
<td>$(</td>
<td>The real group ID (gid) of this process.</td>
</tr>
<tr>
<td>$REAL_GROUP_ID or $GID</td>
<td></td>
</tr>
<tr>
<td>$)</td>
<td>The effective gid of this process.</td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>$EFFECTIVE_GROUP_ID</code> or <code>$EGID</code></td>
<td>Contains the name of the file containing the Perl script being executed.</td>
</tr>
<tr>
<td><code>$0</code></td>
<td>The index of the first element in an array and of the first character in a substring. Default is 0.</td>
</tr>
<tr>
<td><code>$PROGRAM_NAME</code></td>
<td>Returns the version plus patchlevel divided by 1000.</td>
</tr>
<tr>
<td><code>$PERL_VERSION</code></td>
<td>The current value of the debugging flags.</td>
</tr>
<tr>
<td><code>$DEBUGGING</code></td>
<td>Extended error message on some platforms.</td>
</tr>
<tr>
<td><code>$EXTENDED_OS_ERROR</code></td>
<td>The maximum system file descriptor, ordinarily 2.</td>
</tr>
<tr>
<td><code>$SYSTEM_FD_MAX</code></td>
<td>Contains internal compiler hints enabled by certain pragmatic modules.</td>
</tr>
<tr>
<td><code>$^I</code></td>
<td>The current value of the inplace-edit extension. Use undef to disable inplace editing.</td>
</tr>
<tr>
<td><code>$INPLACE_EDIT</code></td>
<td>The contents of $M can be used as an emergency memory pool in case Perl dies with an out-of-memory error. Use of $M requires a special compilation of Perl. See the INSTALL document for</td>
</tr>
<tr>
<td><code>$^M</code></td>
<td></td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>$^O</td>
<td>Contains the name of the operating system that the current Perl binary was compiled for.</td>
</tr>
<tr>
<td>$OSNAME</td>
<td></td>
</tr>
<tr>
<td>$^P</td>
<td>The internal flag that the debugger clears so that it doesn't debug itself.</td>
</tr>
<tr>
<td>$PERLDB</td>
<td></td>
</tr>
<tr>
<td>$^T</td>
<td>The time at which the script began running, in seconds since the epoch.</td>
</tr>
<tr>
<td>$BASETIME</td>
<td></td>
</tr>
<tr>
<td>$^W</td>
<td>The current value of the warning switch, either true or false.</td>
</tr>
<tr>
<td>$WARNING</td>
<td></td>
</tr>
<tr>
<td>$^X</td>
<td>The name that the Perl binary itself was executed as.</td>
</tr>
<tr>
<td>$EXECUTABLE_NAME</td>
<td></td>
</tr>
<tr>
<td>$ARGV</td>
<td>Contains the name of the current file when reading from.</td>
</tr>
</tbody>
</table>

### Global Array Special Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@ARGV</td>
<td>The array containing the command-line arguments intended for the script.</td>
</tr>
<tr>
<td>@INC</td>
<td>The array containing the list of places to look for Perl scripts to be evaluated by the do, require, or use constructs.</td>
</tr>
<tr>
<td>@F</td>
<td>The array into which the input lines are split when the -a command-line switch is given.</td>
</tr>
</tbody>
</table>
Global Hash Special Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%INC</td>
<td>The hash containing entries for the filename of each file that has been included via do or require.</td>
</tr>
<tr>
<td>%ENV</td>
<td>The hash containing your current environment.</td>
</tr>
<tr>
<td>%SIG</td>
<td>The hash used to set signal handlers for various signals.</td>
</tr>
</tbody>
</table>

Global Special Filehandles

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARGV</td>
<td>The special filehandle that iterates over command line filenames in @ARGV. Usually written as the null filehandle in &lt;&gt;.</td>
</tr>
<tr>
<td>STDERR</td>
<td>The special filehandle for standard error in any package.</td>
</tr>
<tr>
<td>STDIN</td>
<td>The special filehandle for standard input in any package.</td>
</tr>
<tr>
<td>STDOUT</td>
<td>The special filehandle for standard output in any package.</td>
</tr>
<tr>
<td>DATA</td>
<td>The special filehandle that refers to anything following the <strong>END</strong> token in the file containing the script. Or, the special filehandle for anything following the <strong>DATA</strong> token in a required file, as long as you’re reading data in the same package <strong>DATA</strong> was found in.</td>
</tr>
<tr>
<td>_ (underscore)</td>
<td>The special filehandle used to cache the information from the last stat, lstat, or file test operator.</td>
</tr>
</tbody>
</table>

Global Special Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>END</strong></td>
<td>Indicates the logical end of your program. Any following text is ignored, but may be read via the</td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>FILE</strong></td>
<td>Represents the filename at the point in your program where it's used. Not interpolated into strings.</td>
</tr>
<tr>
<td><strong>LINE</strong></td>
<td>Represents the current line number. Not interpolated into strings.</td>
</tr>
<tr>
<td><strong>PACKAGE</strong></td>
<td>Represents the current package name at compile time, or undefined if there is no current package. Not interpolated into strings.</td>
</tr>
</tbody>
</table>

**Regular Expression Special Variables**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$digit</td>
<td>Contains the text matched by the corresponding set of parentheses in the last pattern matched. For example, $1 matches whatever was contained in the first set of parentheses in the previous regular expression.</td>
</tr>
<tr>
<td>$&amp;</td>
<td>The string matched by the last successful pattern match.</td>
</tr>
<tr>
<td>$MATCH</td>
<td></td>
</tr>
<tr>
<td>$`</td>
<td>The string preceding whatever was matched by the last successful pattern match.</td>
</tr>
<tr>
<td>$PREMATCH</td>
<td></td>
</tr>
<tr>
<td><code>$</code></td>
<td>The string following whatever was matched by the last successful pattern match.</td>
</tr>
<tr>
<td>$POSTMATCH</td>
<td></td>
</tr>
<tr>
<td>$+</td>
<td>The last bracket matched by the last search pattern. This is useful if you don’t know which of a set of alternative patterns was matched. For example: /Version: (.*)</td>
</tr>
</tbody>
</table>
## Filehandle Special Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td></td>
</tr>
<tr>
<td>$OUTPUT_AUTOFLUSH</td>
<td>The current page number of the currently selected output channel.</td>
</tr>
<tr>
<td>$%</td>
<td>The current page length (printable lines) of the currently selected output channel. Default is 60.</td>
</tr>
<tr>
<td>$FORMAT_PAGE_NUMBER</td>
<td>The number of lines left on the page of the currently selected output channel.</td>
</tr>
<tr>
<td>$=</td>
<td>The name of the current report format for the currently selected output channel. Default is the name of the filehandle.</td>
</tr>
<tr>
<td>$FORMAT_NAME</td>
<td>The name of the current top-of-page format for the currently selected output channel. Default is the name of the filehandle with _TOP appended.</td>
</tr>
</tbody>
</table>

165
Each programmer will, of course, have his or her own preferences in regards to formatting, but there are some general guidelines that will make your programs easier to read, understand, and maintain.

The most important thing is to run your programs under the -w flag at all times. You may turn it off explicitly for particular portions of code via the no warnings pragma or the $^W variable if you must. You should also always run under use strict or know the reason why not. The use sigtrap and even use diagnostics pragmas may also prove useful.

Regarding aesthetics of code lay out, about the only thing Larry cares strongly about is that the closing curly bracket of a multi-line BLOCK should line up with the keyword that started the construct. Beyond that, he has other preferences that aren't so strong:

- 4-column indent.
- Opening curly on same line as keyword, if possible, otherwise line up.
- Space before the opening curly of a multi-line BLOCK.
- One-line BLOCK may be put on one line, including curilies.
- No space before the semicolon.
- Semicolon omitted in "short" one-line BLOCK.
- Space around most operators.
- Space around a "complex" subscript (inside brackets).
- Blank lines between chunks that do different things.
- Uncuddled elses.
- No space between function name and its opening parenthesis.
- Space after each comma.
- Long lines broken after an operator (except and and or).
- Space after last parenthesis matching on current line.
- Line up corresponding items vertically.
- Omit redundant punctuation as long as clarity doesn't suffer.

Here are some other more substantive style issues to think about: Just because you CAN do something a particular way doesn't mean that you SHOULD do it...
that way. Perl is designed to give you several ways to do anything, so consider picking the most readable one. For instance:

```perl
open(FOO,$foo) || die "Can't open $foo: $!";
```

Is better than -

```perl
die "Can't open $foo: $!" unless open(FOO,$foo);
```

Because the second way hides the main point of the statement in a modifier. On the other hand,

```perl
print "Starting analysis\n" if $verbose;
```

Is better than -

```perl
$verbose && print "Starting analysis\n";
```

Because the main point isn't whether the user typed -v or not.

Don't go through silly contortions to exit a loop at the top or the bottom, when Perl provides the last operator so you can exit in the middle. Just "outdent" it a little to make it more visible:

```perl
LINE:
for (;;) {
    statements;
    last LINE if $foo;
    next LINE if /^#/;
    statements;
}
```

Let's see few more important points:

- Don't be afraid to use loop labels--they're there to enhance readability as well as to allow multilevel loop breaks. See the previous example.

- Avoid using grep() (or map()) or 'backticks' in a void context, that is, when you just throw away their return values. Those functions all have return values, so use them. Otherwise use a foreach() loop or the system() function instead.

- For portability, when using features that may not be implemented on every machine, test the construct in an eval to see if it fails. If you know what version or patchlevel a particular feature was implemented, you can test $] ($PERL_VERSION in English) to see if it will be there. The Config
module will also let you interrogate values determined by the Configure program when Perl was installed.

- Choose mnemonic identifiers. If you can't remember what mnemonic means, you've got a problem.

- While short identifiers like $gotit are probably ok, use underscores to separate words in longer identifiers. It is generally easier to read $var_names_like_this than $VarNamesLikeThis, especially for non-native speakers of English. It's also a simple rule that works consistently with VAR_NAMES_LIKE_THIS.

- Package names are sometimes an exception to this rule. Perl informally reserves lowercase module names for "pragma" modules like integer and strict. Other modules should begin with a capital letter and use mixed case, but probably without underscores due to limitations in primitive file systems' representations of module names as files that must fit into a few sparse bytes.

- If you have a really hairy regular expression, use the /x modifier and put in some whitespace to make it look a little less like line noise. Don't use slash as a delimiter when your regex has slashes or backslashes.

- Always check the return codes of system calls. Good error messages should go to STDERR, include which program caused the problem, what the failed system call and arguments were, and (VERY IMPORTANT) should contain the standard system error message for what went wrong. Here's a simple but sufficient example:

```perl
opendir(D, $dir) or die "can't opendir $dir: $!";
```

- Think about reusability. Why waste brainpower on a one-shot when you might want to do something like it again? Consider generalizing your code. Consider writing a module or object class. Consider making your code run cleanly with use strict and use warnings (or -w) in effect. Consider giving away your code. Consider changing your whole world view. Consider... oh, never mind.

- Be consistent.

- Be nice.
A regular expression is a string of characters that defines the pattern or patterns you are viewing. The syntax of regular expressions in Perl is very similar to what you will find within other regular expression supporting programs, such as `sed`, `grep`, and `awk`.

The basic method for applying a regular expression is to use the pattern binding operators `=~` and `!~`. The first operator is a test and assignment operator.

There are three regular expression operators within Perl.

- Match Regular Expression - `m//`
- Substitute Regular Expression - `s///`
- Transliterate Regular Expression - `tr///`

The forward slashes in each case act as delimiters for the regular expression (regex) that you are specifying. If you are comfortable with any other delimiter, then you can use in place of forward slash.

### The Match Operator

The match operator, `m//`, is used to match a string or statement to a regular expression. For example, to match the character sequence "foo" against the scalar `$bar`, you might use a statement like this:

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

$bar = "This is foo and again foo";
if ($bar =~ /foo/){
    print "First time is matching\n";
}else{
    print "First time is not matching\n";
}

$bar = "foo";
if ($bar =~ /foo/){
    print "Second time is matching\n";
}else{
```
print "Second time is not matching\n";
}

When above program is executed, it produces the following result:

First time is matching
Second time is matching

The m// actually works in the same fashion as the q// operator series. You can use any combination of naturally matching characters to act as delimiters for the expression. For example, m{}, m(), and m>< are all valid. So above example can be re-written as follows:

#!/usr/bin/perl

$bar = "This is foo and again foo";
if ($bar =~ m[foo]){
  print "First time is matching\n";
} else{
  print "First time is not matching\n";
}

$bar = "foo";
if ($bar =~ m{foo}){
  print "Second time is matching\n";
} else{
  print "Second time is not matching\n";
}

You can omit m from m// if the delimiters are forward slashes, but for all other delimiters you must use the m prefix.

Note that the entire match expression, that is the expression on the left of =~ or !~ and the match operator, returns true (in a scalar context) if the expression matches. Therefore the statement:

$true = ($foo =~ m/foo/);

will set $true to 1 if $foo matches the regex, or 0 if the match fails. In a list context, the match returns the contents of any grouped expressions. For
example, when extracting the hours, minutes, and seconds from a time string, we can use:

```perl
my ($hours, $minutes, $seconds) = ($time =~ m/\d+:\d+:\d+/);
```

### Match Operator Modifiers

The match operator supports its own set of modifiers. The /g modifier allows for global matching. The /i modifier will make the match case insensitive. Here is the complete list of modifiers:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Makes the match case insensitive.</td>
</tr>
<tr>
<td>m</td>
<td>Specifies that if the string has newline or carriage return characters, the ^ and $ operators will now match against a newline boundary, instead of a string boundary.</td>
</tr>
<tr>
<td>o</td>
<td>Evaluates the expression only once.</td>
</tr>
<tr>
<td>s</td>
<td>Allows use of . to match a newline character.</td>
</tr>
<tr>
<td>x</td>
<td>Allows you to use white space in the expression for clarity.</td>
</tr>
<tr>
<td>g</td>
<td>Globally finds all matches.</td>
</tr>
<tr>
<td>cg</td>
<td>Allows the search to continue even after a global match fails.</td>
</tr>
</tbody>
</table>

### Matching Only Once

There is also a simpler version of the match operator - the ?PATTERN? operator. This is basically identical to the m// operator except that it only matches once within the string you are searching between each call to reset.

For example, you can use this to get the first and last elements within a list:

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

@list = qw/food foosball subeo footnote terfoot canic footbridge/;
```
foreach (@list)
{
    $first = $1 if ?(foo.*)?;
    $last = $1 if /(foo.*)/;
}
print "First: $first, Last: $last\n";

When above program is executed, it produces the following result:

First: food, Last: footbridge

Regular Expression Variables

Regular expression variables include $, which contains whatever the last grouping match matched; $&, which contains the entire matched string; $`, which contains everything before the matched string; and $', which contains everything after the matched string. Following code demonstrates the result:

#!/usr/bin/perl

$string = "The food is in the salad bar";
$string =~ m/foo/; 
print "Before: $`\n";
print "Matched: $&\n";
print "After: $'\n";

When above program is executed, it produces the following result:

Before: The
Matched: foo
After: d is in the salad bar

The Substitution Operator

The substitution operator, s///, is really just an extension of the match operator that allows you to replace the text matched with some new text. The basic form of the operator is:

s/PATTERN/REPLACEMENT/;
The PATTERN is the regular expression for the text that we are looking for. The REPLACEMENT is a specification for the text or regular expression that we want to use to replace the found text with. For example, we can replace all occurrences of dog with cat using the following regular expression:

```
#/user/bin/perl
$string = "The cat sat on the mat";
$string =~ s/cat/dog/;
print "$string\n";
```

When above program is executed, it produces the following result:

```
The dog sat on the mat
```

### Substitution Operator Modifiers

Here is the list of all the modifiers used with substitution operator.

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Makes the match case insensitive.</td>
</tr>
<tr>
<td>m</td>
<td>Specifies that if the string has newline or carriage return characters, the ^ and $ operators will now match against a newline boundary, instead of a string boundary.</td>
</tr>
<tr>
<td>o</td>
<td>Evaluates the expression only once.</td>
</tr>
<tr>
<td>s</td>
<td>Allows use of . to match a newline character.</td>
</tr>
<tr>
<td>x</td>
<td>Allows you to use white space in the expression for clarity.</td>
</tr>
<tr>
<td>g</td>
<td>Replaces all occurrences of the found expression with the replacement text.</td>
</tr>
<tr>
<td>e</td>
<td>Evaluates the replacement as if it were a Perl statement, and uses its return value as the replacement text.</td>
</tr>
</tbody>
</table>
The Translation Operator

Translation is similar, but not identical, to the principles of substitution, but unlike substitution, translation (or transliteration) does not use regular expressions for its search on replacement values. The translation operators are:

```
tr/SEARCHLIST/REPLACEMENTLIST/cds
y/SEARCHLIST/REPLACEMENTLIST/cds
```

The translation replaces all occurrences of the characters in SEARCHLIST with the corresponding characters in REPLACEMENTLIST. For example, using the "The cat sat on the mat." string we have been using in this chapter:

```
#!/user/bin/perl

$string = 'The cat sat on the mat';
$string =~ tr/a/o/;

print "$string\n";
```

When above program is executed, it produces the following result:

```
The cot sot on the mot.
```

Standard Perl ranges can also be used, allowing you to specify ranges of characters either by letter or numerical value. To change the case of the string, you might use the following syntax in place of the \texttt{uc} function.

```
$string =~ tr/a-z/A-Z/;
```

Translation Operator Modifiers

Following is the list of operators related to translation.

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Complements SEARCHLIST.</td>
</tr>
<tr>
<td>d</td>
<td>Deletes found but unreplaced characters.</td>
</tr>
<tr>
<td>s</td>
<td>Squashes duplicate replaced characters.</td>
</tr>
</tbody>
</table>
The `/d` modifier deletes the characters matching SEARCHLIST that do not have a corresponding entry in REPLACEMENTLIST. For example:

```perl
#!/usr/bin/perl
$string = 'the cat sat on the mat.';
$string =~ tr/a-zA-Z/b/d;
print "$string\n";
```

When above program is executed, it produces the following result:

```
  b b  b.
```

The last modifier, `/s`, removes the duplicate sequences of characters that were replaced, so:

```perl
#!/usr/bin/perl
$string = 'food';
$string = 'food';
$string =~ tr/a-zA-Z/a-z/s;
print "$string\n";
```

When above program is executed, it produces the following result:

```
fod
```

**More Complex Regular Expressions**

You don't just have to match on fixed strings. In fact, you can match on just about anything you could dream of by using more complex regular expressions. Here's a quick cheat sheet:

Following table lists the regular expression syntax that is available in Python.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>^</code></td>
<td>Matches beginning of line.</td>
</tr>
<tr>
<td>Character</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>$</td>
<td>Matches end of line.</td>
</tr>
<tr>
<td>.</td>
<td>Matches any single character except newline. Using m option allows it to match newline as well.</td>
</tr>
<tr>
<td>[...]</td>
<td>Matches any single character in brackets.</td>
</tr>
<tr>
<td>[^...]</td>
<td>Matches any single character not in brackets.</td>
</tr>
<tr>
<td></td>
<td>Matches 0 or more occurrences of preceding expression.</td>
</tr>
<tr>
<td>+</td>
<td>Matches 1 or more occurrence of preceding expression.</td>
</tr>
<tr>
<td>?</td>
<td>Matches 0 or 1 occurrence of preceding expression.</td>
</tr>
<tr>
<td>{ n}</td>
<td>Matches exactly n number of occurrences of preceding expression.</td>
</tr>
<tr>
<td>{ n,}</td>
<td>Matches n or more occurrences of preceding expression.</td>
</tr>
<tr>
<td>{ n, m}</td>
<td>Matches at least n and at most m occurrences of preceding expression.</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>\w</td>
<td>Matches word characters.</td>
</tr>
<tr>
<td>\W</td>
<td>Matches nonword characters.</td>
</tr>
<tr>
<td>\s</td>
<td>Matches whitespace. Equivalent to [\t\n\r\f].</td>
</tr>
<tr>
<td>\S</td>
<td>Matches nonwhitespace.</td>
</tr>
<tr>
<td>\d</td>
<td>Matches digits. Equivalent to [0-9].</td>
</tr>
<tr>
<td>\D</td>
<td>Matches nondigits.</td>
</tr>
<tr>
<td>\A</td>
<td>Matches beginning of string.</td>
</tr>
<tr>
<td>\Z</td>
<td>Matches end of string. If a newline exists, it matches just before newline.</td>
</tr>
<tr>
<td>\z</td>
<td>Matches end of string.</td>
</tr>
<tr>
<td>\G</td>
<td>Matches point where last match finished.</td>
</tr>
<tr>
<td>\b</td>
<td>Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.</td>
</tr>
<tr>
<td>\B</td>
<td>Matches nonword boundaries.</td>
</tr>
<tr>
<td>\n, \t, etc.</td>
<td>Matches newlines, carriage returns, tabs, etc.</td>
</tr>
<tr>
<td>\1...\9</td>
<td>Matches nth grouped subexpression.</td>
</tr>
<tr>
<td>\10</td>
<td>Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.</td>
</tr>
<tr>
<td>[aeiou]</td>
<td>Matches a single character in the given set</td>
</tr>
<tr>
<td>[^aeiou]</td>
<td>Matches a single character outside the given set</td>
</tr>
</tbody>
</table>

The ^ metacharacter matches the beginning of the string and the $ metasymbol matches the end of the string. Here are some brief examples.

```perl
# nothing in the string (start and end are adjacent)
/^$/

# a three digits, each followed by a whitespace
# character (eg "3 4 5 ")
/\d\s\{3\}/

# matches a string in which every
# odd-numbered letter is a (eg "abacadaf")
```
 /(a.+)/

 # string starts with one or more digits
 /\^\d+/

 # string that ends with one or more digits
 /\d+$/

 Let's have a look at another example.

#!/usr/bin/perl

$string = "Cats go Catatonic\nWhen given Catnip";
($start) = ($string =~ /\A(.*?)/);
@lines = $string =~ /^(.*)/gm;
print "First word: $start\n";"Line starts: @lines\n";

When above program is executed, it produces the following result:

First word: Cats
Line starts: Cats When

Matching Boundaries

The \b matches at any word boundary, as defined by the difference between the \w class and the \W class. Because \w includes the characters for a word, and \W the opposite, this normally means the termination of a word. The \B assertion matches any position that is not a word boundary. For example:

/\bcat\b/ # Matches 'the cat sat' but not 'cat on the mat'
/\Bcat\B/ # Matches 'verification' but not 'the cat on the mat'
/\bcat\B/ # Matches 'catatonic' but not 'polecat'
/\Bcat\b/ # Matches 'polecat' but not 'catatonic'
Selecting Alternatives

The | character is just like the standard or bitwise OR within Perl. It specifies alternate matches within a regular expression or group. For example, to match "cat" or "dog" in an expression, you might use this:

```perl
if ($string =~ /cat|dog/)
```

You can group individual elements of an expression together in order to support complex matches. Searching for two people’s names could be achieved with two separate tests, like this:

```perl
if (($string =~ /Martin Brown/) || ($string =~ /Sharon Brown/))
```

This could be written as follows

```perl
if ($string =~ /(Martin|Sharon) Brown/)
```

Grouping Matching

From a regular-expression point of view, there is no difference between except, perhaps, that the former is slightly clearer.

```perl
$string =~ /([^\s]+)\s+([^\s]+)/;

and

$string =~ /\s+([^\s]+)\s+([^\s]+)/;
```

However, the benefit of grouping is that it allows us to extract a sequence from a regular expression. Groupings are returned as a list in the order in which they appear in the original. For example, in the following fragment we have pulled out the hours, minutes, and seconds from a string.

```perl
my ($hours, $minutes, $seconds) = ($time =~ m/([\d]+):([\d]+):([\d]+)/);
```

As well as this direct method, matched groups are also available within the special $x variables, where x is the number of the group within the regular expression. We could therefore rewrite the preceding example as follows:

```perl
#!/usr/bin/perl
```
$time = "12:05:30";

$time =~ m/(\d+):(\d+):(\d+)/;
my ($hours, $minutes, $seconds) = ($1, $2, $3);

print "Hours : $hours, Minutes: $minutes, Second: $seconds\n"

When above program is executed, it produces the following result:

Hours : 12, Minutes: 05, Second: 30

When groups are used in substitution expressions, the $x syntax can be used in the replacement text. Thus, we could reformat a date string using this:

#!/usr/bin/perl
$date = '03/26/1999';
$date =~ s#(\d+)/((\d+)/((\d+))$3/$1/$2#;
print "$date\n";

When above program is executed, it produces the following result:

1999/03/26

The \G Assertion

The \G assertion allows you to continue searching from the point where the last match occurred. For example, in the following code, we have used \G so that we can search to the correct position and then extract some information, without having to create a more complex, single regular expression:

#!/usr/bin/perl
$string = "The time is: 12:31:02 on 4/12/00";

$string =~ /s+/g;
($time) = ($string =~ /\G(\d+:\d+:\d+)/);
$string =~ .+\s+/g;
($date) = ($string =~ m{\G(\d+/\d+/\d+)})

print "Time: $time, Date: $date\n"

When above program is executed, it produces the following result:

| Time: 12:31:02, Date: 4/12/00 |

The \G assertion is actually just the metasymbol equivalent of the pos function, so between regular expression calls you can continue to use pos, and even modify the value of pos (and therefore \G) by using pos as an lvalue subroutine.

### Regular-expression Examples

#### Literal Characters

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perl</td>
<td>Match &quot;Perl&quot;.</td>
</tr>
</tbody>
</table>

#### Character Classes

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Pp]ython</td>
<td>Matches &quot;Python&quot; or &quot;python&quot;</td>
</tr>
<tr>
<td>rub[ye]</td>
<td>Matches &quot;ruby&quot; or &quot;rube&quot;</td>
</tr>
<tr>
<td>[aeiou]</td>
<td>Matches any one lowercase vowel</td>
</tr>
<tr>
<td>[0-9]</td>
<td>Matches any digit; same as [0123456789]</td>
</tr>
<tr>
<td>[a-z]</td>
<td>Matches any lowercase ASCII letter</td>
</tr>
<tr>
<td>[A-Z]</td>
<td>Matches any uppercase ASCII letter</td>
</tr>
<tr>
<td>[a-zA-Z0-9]</td>
<td>Matches any of the above</td>
</tr>
</tbody>
</table>
[^aeiou] Matches anything other than a lowercase vowel
[^0-9] Matches anything other than a digit

### Special Character Classes

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Matches any character except newline</td>
</tr>
<tr>
<td>\d</td>
<td>Matches a digit: [0-9]</td>
</tr>
<tr>
<td>\D</td>
<td>Matches a nondigit: [^0-9]</td>
</tr>
<tr>
<td>\s</td>
<td>Matches a whitespace character: [ \t\r\n\f]</td>
</tr>
<tr>
<td>\S</td>
<td>Matches nonwhitespace: [^ \t\r\n\f]</td>
</tr>
<tr>
<td>\w</td>
<td>Matches a single word character: [A-Za-z0-9_]</td>
</tr>
<tr>
<td>\W</td>
<td>Matches a nonword character: [^A-Za-z0-9_]</td>
</tr>
</tbody>
</table>

### Repetition Cases

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ruby?</td>
<td>Matches &quot;rub&quot; or &quot;ruby&quot;: the y is optional</td>
</tr>
<tr>
<td>ruby*</td>
<td>Matches &quot;rub&quot; plus 0 or more ys</td>
</tr>
<tr>
<td>ruby+</td>
<td>Matches &quot;rub&quot; plus 1 or more ys</td>
</tr>
<tr>
<td>\d{3}</td>
<td>Matches exactly 3 digits</td>
</tr>
<tr>
<td>\d{3,}</td>
<td>Matches 3 or more digits</td>
</tr>
</tbody>
</table>
\d{3,5} \nMatches 3, 4, or 5 digits

**Nongreedy Repetition**

This matches the smallest number of repetitions:

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;.*&gt;</td>
<td>Greedy repetition: matches &quot;&lt;python&gt;perl&gt;&quot;</td>
</tr>
<tr>
<td>&lt;.*?&gt;</td>
<td>Nongreedy: matches &quot;&lt;python&gt;&quot; in &quot;&lt;python&gt;perl&gt;&quot;</td>
</tr>
</tbody>
</table>

**Grouping with Parentheses**

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\D\d+</td>
<td>No group: + repeats \d</td>
</tr>
<tr>
<td>(\D\d)+</td>
<td>Grouped: + repeats \D\d pair</td>
</tr>
<tr>
<td>([Pp]ython(, )?)+</td>
<td>Match &quot;Python&quot;, &quot;Python, python, python&quot;, etc.</td>
</tr>
</tbody>
</table>

**Backreferences**

This matches a previously matched group again:

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>([Pp])ython&amp;\1ails</td>
<td>Matches python&amp;pails or Python&amp;Pails</td>
</tr>
<tr>
<td>(&quot;&quot;\1)*\1</td>
<td>Single or double-quoted string. \1 matches whatever the 1st group matched. \2 matches whatever the 2nd group matched, etc.</td>
</tr>
</tbody>
</table>

**Alternatives**

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
</table>

183
python|perl | Matches "python" or "perl"
rub(y|le)) | Matches "ruby" or "ruble"
Python(!+|\?) | "Python" followed by one or more ! or one ?

**Anchors**
This need to specify match positions.

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^Python</td>
<td>Matches &quot;Python&quot; at the start of a string or internal line</td>
</tr>
<tr>
<td>Python$</td>
<td>Matches &quot;Python&quot; at the end of a string or line</td>
</tr>
<tr>
<td>\APython</td>
<td>Matches &quot;Python&quot; at the start of a string</td>
</tr>
<tr>
<td>Python\Z</td>
<td>Matches &quot;Python&quot; at the end of a string</td>
</tr>
<tr>
<td>\bPython\b</td>
<td>Matches &quot;Python&quot; at a word boundary</td>
</tr>
<tr>
<td>\brub\B</td>
<td>\B is nonword boundary: match &quot;rub&quot; in &quot;rube&quot; and &quot;ruby&quot; but not alone</td>
</tr>
<tr>
<td>Python(?=!))</td>
<td>Matches &quot;Python&quot;, if followed by an exclamation point</td>
</tr>
<tr>
<td>Python(?!!))</td>
<td>Matches &quot;Python&quot;, if not followed by an exclamation point</td>
</tr>
</tbody>
</table>

**Special Syntax with Parentheses**

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(?#comment)</td>
<td>Matches &quot;R&quot;. All the rest is a comment</td>
</tr>
<tr>
<td>R(?i)uby</td>
<td>Case-insensitive while matching &quot;uby&quot;</td>
</tr>
<tr>
<td>\textbf{Perl}</td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td>R(?i:uby)</td>
<td>Same as above</td>
</tr>
<tr>
<td>rub(?:\y</td>
<td>le))</td>
</tr>
</tbody>
</table>
Using sendmail Utility

Sending a Plain Message

If you are working on Linux/Unix machine then you can simply use sendmail utility inside your Perl program to send email. Here is a sample script that can send an email to a given email ID. Just make sure the given path for sendmail utility is correct. This may be different for your Linux/Unix machine.

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

$to = 'abcd@gmail.com';
(from = 'webmaster@yourdomain.com';
$subject = 'Test Email';
$message = 'This is test email sent by Perl Script';

open(MAIL, "|/usr/sbin/sendmail -t");

# Email Header
print MAIL "To: $to\n";
print MAIL "From: $from\n";
print MAIL "Subject: $subject\n";
# Email Body
print MAIL $message;

close(MAIL);
print "Email Sent Successfully\n";
```

Actually, the above script is a client email script, which will draft email and submit to the server running locally on your Linux/Unix machine. This script will not be responsible for sending email to actual destination. So you have to make sure email server is properly configured and running on your machine to send email to the given email ID.
## Sending an HTML Message

If you want to send HTML formatted email using sendmail, then you simply need to add **Content-type: text/html** in the header part of the email as follows:

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

$to = 'abcd@gmail.com';
$from = 'webmaster@yourdomain.com';
$subject = 'Test Email';
$message = '<h1>This is test email sent by Perl Script</h1>';

open(MAIL, "|/usr/sbin/sendmail -t");

# Email Header
print MAIL "To: $to\n";
print MAIL "From: $from\n";
print MAIL "Subject: $subject\n\n";
print MAIL "Content-type: text/html\n";
# Email Body
print MAIL $message;

close(MAIL);
print "Email Sent Successfully\n";
```

## Using MIME::Lite Module

If you are working on windows machine, then you will not have access on sendmail utility. But you have alternate to write your own email client using MIME::Lite perl module. You can download this module from [MIME-Lite-3.01.tar.gz](http://example.com) and install it on your either machine Windows or Linux/Unix. To install it follow the simple steps:

```
$tar xvfz MIME-Lite-3.01.tar.gz
$cd MIME-Lite-3.01
$perl Makefile.PL
$make
```
That's it and you will have MIME::Lite module installed on your machine. Now you are ready to send your email with simple scripts explained below.

### Sending a Plain Message

Now following is a script which will take care of sending email to the given email ID:

```perl
#!/usr/bin/perl
use MIME::Lite;

$to = 'abcd@gmail.com';
$cc = 'efgh@mail.com';
$from = 'webmaster@yourdomain.com';
$subject = 'Test Email';
$message = 'This is test email sent by Perl Script';

$msg = MIME::Lite->new(
    From => $from,
    To => $to,
    Cc => $cc,
    Subject => $subject,
    Data => $message
);

$msg->send;
print "Email Sent Successfully\n";
```

### Sending an HTML Message

If you want to send HTML formatted email using sendmail, then you simply need to add `Content-type: text/html` in the header part of the email. Following is the script, which will take care of sending HTML formatted email:

```perl
#!/usr/bin/perl
use MIME::Lite;
```
$to = 'abcd@gmail.com';
$cc = 'efgh@mail.com';
from = 'webmaster@yourdomain.com';
$subject = 'Test Email';
$message = '<h1>This is test email sent by Perl Script</h1>';  

$msg = MIME::Lite->new(  
    From => $from,  
    To => $to,  
    Cc => $cc,  
    Subject => $subject,  
    Data => $message  
);

$msg->attr("content-type" => "text/html");
$msg->send;
print "Email Sent Successfully\n";

Sending an Attachment

If you want to send an attachment, then following script serves the purpose:

#!/usr/bin/perl
use MIME::Lite;

$to = 'abcd@gmail.com';
$cc = 'efgh@mail.com';
from = 'webmaster@yourdomain.com';
$subject = 'Test Email';
$message = 'This is test email sent by Perl Script';

$msg = MIME::Lite->new(  
    From => $from,  
    To => $to,  
    Cc => $cc,  
    Subject => $subject,  
    Data => $message  
);
You can attach as many files as you like in your email using attach() method.

### Using SMTP Server

If your machine is not running an email server then you can use any other email server available at the remote location. But to use any other email server you will need to have an id, its password, URL, etc. Once you have all the required information, you simple need to provide that information in `send()` method as follows:

```perl
$msg->send('smtp', "smtp.myisp.net", AuthUser=>"id", AuthPass=>"password");
```

You can contact your email server administrator to have the above used information and if a user id and password is not already available then your administrator can create it in minutes.
Part 2: Perl – Advanced Topics
What is a Socket?

Socket is a Berkeley UNIX mechanism of creating a virtual duplex connection between different processes. This was later ported on to every known OS enabling communication between systems across geographical location running on different OS software. If not for the socket, most of the network communication between systems would never ever have happened.

Taking a closer look; a typical computer system on a network receives and sends information as desired by the various applications running on it. This information is routed to the system, since a unique IP address is designated to it. On the system, this information is given to the relevant applications, which listen on different ports. For example an internet browser listens on port 80 for information received from the web server. Also we can write our custom applications which may listen and send/receive information on a specific port number.

For now, let's sum up that a socket is an IP address and a port, enabling connection to send and receive data over a network.

To explain above mentioned socket concept we will take an example of Client - Server Programming using Perl. To complete a client server architecture we would have to go through the following steps:

To Create a Server

- Create a socket using `socket` call.
- Bind the socket to a port address using `bind` call.
- Listen to the socket at the port address using `listen` call.
- Accept client connections using `accept` call.

To Create a Client

- Create a socket with `socket` call.
- Connect (the socket) to the server using `connect` call.

Following diagram shows the complete sequence of the calls used by Client and Server to communicate with each other:
Server Side Socket Calls

The socket() call

The socket() call is the first call in establishing a network connection is creating a socket. This call has the following syntax:

```perl
socket( SOCKET, DOMAIN, TYPE, PROTOCOL );
```

The above call creates a SOCKET and other three arguments are integers which should have the following values for TCP/IP connections.

- **DOMAIN** should be PF_INET. It's probable 2 on your computer.
- **TYPE** should be SOCK_STREAM for TCP/IP connection.
- **PROTOCOL** should be `(getprotobyname('tcp'))[2]`. It is the particular protocol such as TCP to be spoken over the socket.

So socket function call issued by the server will be something like this:

```perl
use Socket  # This defines PF_INET and SOCK_STREAM

socket(SOCKET,PF_INET,SOCK_STREAM,(getprotobyname('tcp'))[2]);
```
The bind() call

The sockets created by socket() call are useless until they are bound to a hostname and a port number. Server uses the following `bind()` function to specify the port at which they will be accepting connections from the clients.

```perl
bind( SOCKET, ADDRESS );
```

Here SOCKET is the descriptor returned by socket() call and ADDRESS is a socket address (for TCP/IP) containing three elements:

- The address family (For TCP/IP, that's AF_INET, probably 2 on your system).
- The port number (for example 21).
- The internet address of the computer (for example 10.12.12.168).

As the bind() is used by a server, which does not need to know its own address so the argument list looks like this:

```perl
use Socket # This defines PF_INET and SOCK_STREAM

$port = 12345; # The unique port used by the sever to listen requests
$server_ip_address = "10.12.12.168";
bind( SOCKET, pack_sockaddr_in($port, inet_aton($server_ip_address)))
  or die "Can't bind to port $port! \n";
```

The `or die` clause is very important because if a server dies without outstanding connections, the port won't be immediately reusable unless you use the option `SO_REUSEADDR` using `setsockopt()` function. Here `pack_sockaddr_in()` function is being used to pack the Port and IP address into binary format.

The listen() call

If this is a server program, then it is required to issue a call to `listen()` on the specified port to listen, i.e., wait for the incoming requests. This call has the following syntax:

```perl
listen( SOCKET, QUEUESIZE );
```

The above call uses SOCKET descriptor returned by socket() call and QUEUESIZE is the maximum number of outstanding connection request allowed simultaneously.
The accept() call

If this is a server program then it is required to issue a call to the `access()` function to accept the incoming connections. This call has the following syntax:

```perl
global_socket = accept( NEW_SOCKET, SOCKET );
```

The accept call receive SOCKET descriptor returned by socket() function and upon successful completion, a new socket descriptor NEW_SOCKET is returned for all future communication between the client and the server. If access() call fails, then it returns FALSE which is defined in Socket module which we have used initially.

Generally, accept() is used in an infinite loop. As soon as one connection arrives the server either creates a child process to deal with it or serves it himself and then goes back to listen for more connections.

```perl
while(1) {
    accept( NEW_SOCKET, SOCKET );
    .......
}
```

Now all the calls related to server are over and let us see a call which will be required by the client.

Client Side Socket Calls

The connect() call

If you are going to prepare client program, then first you will use `socket()` call to create a socket and then you would have to use `connect()` call to connect to the server. You already have seen socket() call syntax and it will remain similar to server socket() call, but here is the syntax for `connect()` call:

```perl
global_socket = connect( SOCKET, ADDRESS );
```

Here SOCKET is the socket descriptor returned by socket() call issued by the client and ADDRESS is a socket address similar to bind call, except that it contains the IP address of the remote server.

```perl
$port = 21;       # For example, the ftp port
$server_ip_address = "10.12.12.168";
connect( SOCKET, pack_sockaddr_in($port, inet_aton($server_ip_address)))
```
If you connect to the server successfully, then you can start sending your commands to the server using SOCKET descriptor, otherwise your client will come out by giving an error message.

**Client - Server Example**

Following is a Perl code to implement a simple client-server program using Perl socket. Here server listens for incoming requests and once connection is established, it simply replies *Smile from the server*. The client reads that message and print on the screen. Let's see how it has been done, assuming we have our server and client on the same machine.

**Script to Create a Server**

```perl
#!/usr/bin/perl -w

# Filename : server.pl

use strict;
use Socket;

# use port 7890 as default
my $port = shift || 7890;
my $proto = getprotobyname('tcp');
my $server = "localhost"; # Host IP running the server

# create a socket, make it reusable
socket(SOCKET, PF_INET, SOCK_STREAM, $proto)
    or die "Can't open socket $!

setsockopt(SOCKET, SOL_SOCKET, SO_REUSEADDR, 1)
    or die "Can't set socket option to SO_REUSEADDR $!

# bind to a port, then listen
bind( SOCKET, pack_sockaddr_in($port, inet_aton($server)))
    or die "Can't bind to port $port! 
```

196
Perl

```perl
listen(SOCKET, 5) or die "listen: $!";
print "SERVER started on port $port\n";

# accepting a connection
my $client_addr;
while ($client_addr = accept(NEW_SOCKET, SOCKET)) {
    # send them a message, close connection
    my $name = gethostbyaddr($client_addr, AF_INET);
    print NEW_SOCKET "Smile from the server";
    print "Connection received from $name\n";
    close NEW_SOCKET;
}
```

To run the server in background mode issue the following command on Unix prompt:

```
$perl server.pl&
```

**Script to Create a Client**

```
!/usr/bin/perl -w
# Filename : client.pl

use strict;
use Socket;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 7890;
my $server = "localhost";  # Host IP running the server

# create the socket, connect to the port
socket(SOCKET,PF_INET,SOCK_STREAM,(getprotobynumber('tcp'))[2])
or die "Can't create a socket $!\n";
connect( SOCKET, pack_sockaddr_in($port, inet_aton($server)))
```
or die "Can't connect to port $port! \n";

my $line;
while ($line = <SOCKET>) {
    print "$line\n";
}
close SOCKET or die "close: $!";

Now let's start our client at the command prompt, which will connect to the server and read message sent by the server and displays the same on the screen as follows:

$perl client.pl
Smile from the server

**NOTE:** If you are giving the actual IP address in dot notation, then it is recommended to provide IP address in the same format in both client as well as server to avoid any confusion.
We have already studied references in Perl and Perl anonymous arrays and hashes. Object Oriented concept in Perl is very much based on references and anonymous array and hashes. Let's start learning basic concepts of Object Oriented Perl.

**Object Basics**

There are three main terms, explained from the point of view of how Perl handles objects. The terms are object, class, and method.

- An **object** within Perl is merely a reference to a data type that knows what class it belongs to. The object is stored as a reference in a scalar variable. Because a scalar only contains a reference to the object, the same scalar can hold different objects in different classes.

- A **class** within Perl is a package that contains the corresponding methods required to create and manipulate objects.

- A **method** within Perl is a subroutine, defined with the package. The first argument to the method is an object reference or a package name, depending on whether the method affects the current object or the class.

Perl provides a **bless()** function, which is used to return a reference which ultimately becomes an object.

**Defining a Class**

It is very simple to define a class in Perl. A class is corresponding to a Perl Package in its simplest form. To create a class in Perl, we first build a package.

A package is a self-contained unit of user-defined variables and subroutines, which can be re-used over and over again.

Perl Packages provide a separate namespace within a Perl program which keeps subroutines and variables independent from conflicting with those in other packages.

To declare a class named Person in Perl we do:

```perl
package Person;
```

The scope of the package definition extends to the end of the file, or until another package keyword is encountered.
Creating and Using Objects

To create an instance of a class (an object) we need an object constructor. This constructor is a method defined within the package. Most programmers choose to name this object constructor method new, but in Perl you can use any name.

You can use any kind of Perl variable as an object in Perl. Most Perl programmers choose either references to arrays or hashes.

Let's create our constructor for our Person class using a Perl hash reference. When creating an object, you need to supply a constructor, which is a subroutine within a package that returns an object reference. The object reference is created by blessing a reference to the package's class. For example:

```perl
package Person;
sub new {
    my $class = shift;
    my $self = {
        _firstName => shift,
        _lastName  => shift,
        _ssn       => shift,
    };
    # Print all the values just for clarification.
    print "First Name is $self->{_firstName}\n";
    print "Last Name is $self->{_lastName}\n";
    print "SSN is $self->{_ssn}\n";
    bless $self, $class;
    return $self;
}
```

Now Let us see how to create an Object.

```perl
$object = new Person( "Mohammad", "Saleem", 23234345);
```

You can use simple hash in your constructor if you don't want to assign any value to any class variable. For example:

```perl
package Person;
sub new {

}
my $class = shift;
my $self = {};
bless $self, $class;
return $self;
}

Defining Methods

Other object-oriented languages have the concept of security of data to prevent a programmer from changing an object data directly and they provide accessor methods to modify object data. Perl does not have private variables but we can still use the concept of helper methods to manipulate object data.

Lets define a helper method to get person’s first name:

```perl
sub getFirstName {
    return $self->{_firstName};
}
```

Another helper function to set person’s first name:

```perl
sub setFirstName {
    my ( $self, $firstName ) = @_;
    $self->{_firstName} = $firstName if defined($firstName);
    return $self->{_firstName};
}
```

Now lets have a look into complete example: Keep Person package and helper functions into Person.pm file.

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

package Person;

sub new {
    my $class = shift;
    my $self = {
        _firstName => shift,
    }
Now let's make use of Person object in employee.pl file as follows:

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

use Person;

$object = new Person("Mohammad", "Saleem", 23234345);
# Get first name which is set using constructor.
$firstName = $object->getFirstName();

print "Before Setting First Name is : $firstName\n";

# Now Set first name using helper function.
```
$object->setFirstName("Mohd.");

# Now get first name set by helper function.
$firstName = $object->getFirstName();
print "Before Setting First Name is: $firstName\n";

When we execute above program, it produces the following result:

First Name is Mohammad
Last Name is Saleem
SSN is 23234345
Before Setting First Name is: Mohammad
Before Setting First Name is: Mohd.

Inheritance

Object-oriented programming has very good and useful concept called inheritance. Inheritance simply means that properties and methods of a parent class will be available to the child classes. So you don't have to write the same code again and again, you can just inherit a parent class.

For example, we can have a class Employee, which inherits from Person. This is referred to as an "isa" relationship because an employee is a person. Perl has a special variable, @ISA, to help with this. @ISA governs (method) inheritance.

Following are the important points to be considered while using inheritance:

- Perl searches the class of the specified object for the given method or attribute, i.e., variable.
- Perl searches the classes defined in the object class's @ISA array.
- If no method is found in steps 1 or 2, then Perl uses an AUTOLOAD subroutine, if one is found in the @ISA tree.
- If a matching method still cannot be found, then Perl searches for the method within the UNIVERSAL class (package) that comes as part of the standard Perl library.
- If the method still has not found, then Perl gives up and raises a runtime exception.

So to create a new Employee class that will inherit methods and attributes from our Person class, we simply code as follows: Keep this code into Employee.pm.

#!/usr/bin/perl
package Employee;
use Person;
use strict;
our @ISA = qw(Person);  # inherits from Person

Now Employee Class has all the methods and attributes inherited from Person class and you can use them as follows: Use main.pl file to test it:

#!/usr/bin/perl

use Employee;

$object = new Employee( "Mohammad", "Saleem", 23234345);
# Get first name which is set using constructor.
$firstName = $object->getFirstName();

print "Before Setting First Name is : $firstName\n";

# Now Set first name using helper function.
$object->setFirstName( "Mohd." );

# Now get first name set by helper function.
$firstName = $object->getFirstName();
print "After Setting First Name is : $firstName\n";

When we execute above program, it produces the following result:

First Name is Mohammad
Last Name is Saleem
SSN is 23234345
Before Setting First Name is : Mohammad
Before Setting First Name is : Mohd.
Method Overriding

The child class Employee inherits all the methods from the parent class Person. But if you would like to override those methods in your child class then you can do it by giving your own implementation. You can add your additional functions in child class or you can add or modify the functionality of an existing methods in its parent class. It can be done as follows: modify Employee.pm file.

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

package Employee;
use Person;
use strict;
our @ISA = qw(Person);  # inherits from Person

# Override constructor
sub new {
    my ($class) = @_;

    # Call the constructor of the parent class, Person.
    my $self = $class->SUPER::new( $_[1], $_[2], $_[3] );
    # Add few more attributes
    $self->{_id} = undef;
    $self->{_title} = undef;
    bless $self, $class;
    return $self;
}

# Override helper function
sub getFirstName {
    my( $self ) = @_;  
    # This is child class function.
    print "This is child class helper function\n";
    return $self->{_firstName};
}
```
Now let's again try to use Employee object in our main.pl file and execute it.

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

use Employee;

$object = new Employee("Mohammad", "Saleem", 23234345);
# Get first name which is set using constructor.
$firstName = $object->getFirstName();

print "Before Setting First Name is : $firstName\n";

# Now Set first name using helper function.
$object->setFirstName("Mohd.");

# Now get first name set by helper function.
$firstName = $object->getFirstName();
print "After Setting First Name is : $firstName\n";
```

When we execute above program, it produces the following result:

```
First Name is Mohammad
```
Last Name is Saleem  
SSN is 23234345  
This is child class helper function  
Before Setting First Name is : Mohammad  
This is child class helper function  
After Setting First Name is : Mohd.

Default Autoloading

Perl offers a feature which you would not find in any other programming languages: a default subroutine. Which means, if you define a function called AUTOLOAD(), then any calls to undefined subroutines will call AUTOLOAD() function automatically. The name of the missing subroutine is accessible within this subroutine as $AUTOLOAD.

Default autoloading functionality is very useful for error handling. Here is an example to implement AUTOLOAD, you can implement this function in your own way.

```
sub AUTOLOAD
{
    my $self = shift;
    my $type = ref ($self) || croak "$self is not an object";
    my $field = $AUTOLOAD;
    $field =~ s/.*://;
    unless (exists $self->{$field})
    {
        croak "$field does not exist in object/class $type";
    }
    if (@__)
    {
        return $self->($name) = shift;
    }
    else
    {
        return $self->($name);
    }
}
```
Destructors and Garbage Collection

If you have programmed using object oriented programming before, then you will be aware of the need to create a destructor to free the memory allocated to the object when you have finished using it. Perl does this automatically for you as soon as the object goes out of scope.

In case you want to implement your destructor, which should take care of closing files or doing some extra processing then you need to define a special method called DESTROY. This method will be called on the object just before Perl frees the memory allocated to it. In all other respects, the DESTROY method is just like any other method, and you can implement whatever logic you want inside this method.

A destructor method is simply a member function (subroutine) named DESTROY, which will be called automatically in following cases:

- When the object reference's variable goes out of scope.
- When the object reference's variable is undef-ed.
- When the script terminates
- When the perl interpreter terminates

For Example, you can simply put the following method DESTROY in your class:

```perl
package MyClass;
...
sub DESTROY
{
    print "MyClass::DESTROY called\n";
}
```

Object Oriented Perl Example

Here is another nice example, which will help you to understand Object Oriented Concepts of Perl. Put this source code into any perl file and execute it.

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

# Following is the implementation of simple Class.
package MyClass;
```
sub new
{
    print "MyClass::new called\n";
    my $type = shift;       # The package/type name
    my $self = {};          # Reference to empty hash
    return bless $self, $type;
}

sub DESTROY
{
    print "MyClass::DESTROY called\n";
}

sub MyMethod
{
    print "MyClass::MyMethod called!\n";
}

# Following is the implementation of Inheritance.
package MySubClass;

@ISA = qw( MyClass );

sub new
{
    print "MySubClass::new called\n";
    my $type = shift;       # The package/type name
    my $self = MyClass->new; # Reference to empty hash
    return bless $self, $type;
}
sub DESTROY
{
    print "MySubClass::DESTROY called\n";
}

sub MyMethod
{
    my $self = shift;
    $self-&gt;SUPER::MyMethod();
    print " MySubClass::MyMethod called!\n";
}

# Here is the main program using above classes.
package main;

print "Invoke MyClass method\n";
$myObject = MyClass-&gt;new();
$myObject-&gt;MyMethod();

print "Invoke MySubClass method\n";
$myObject2 = MySubClass-&gt;new();
$myObject2-&gt;MyMethod();

print "Create a scoped object\n";
{
    my $myObject2 = MyClass-&gt;new();
}
# Destructor is called automatically here

print "Create and undef an object\n";
$myObject3 = MyClass-&gt;new();
undef $myObject3;

print "Fall off the end of the script...\n";
# Remaining destructors are called automatically here

When we execute above program, it produces the following result:

Invoke MyClass method
MyClass::new called
MyClass::MyMethod called!
Invoke MySubClass method
MySubClass::new called
MyClass::new called
MyClass::MyMethod called!
    MySubClass::MyMethod called!
Create a scoped object
MyClass::new called
MyClass::DESTROY called
Create and undef an object
MyClass::new called
MyClass::DESTROY called
Fall off the end of the script...
MyClass::DESTROY called
MySubClass::DESTROY called
This chapter teaches you how to access a database inside your Perl script. Starting from Perl 5 has become very easy to write database applications using DBI module. DBI stands for **Database Independent Interface** for Perl, which means DBI provides an abstraction layer between the Perl code and the underlying database, allowing you to switch database implementations really easily.

The DBI is a database access module for the Perl programming language. It provides a set of methods, variables, and conventions that provide a consistent database interface, independent of the actual database being used.

### Architecture of a DBI Application

DBI is independent of any database available in backend. You can use DBI whether you are working with Oracle, MySQL or Informix, etc. This is clear from the following architecture diagram.

![Architecture of a DBI Application Diagram](image)

Here DBI is responsible of taking all SQL commands through the API, (i.e., Application Programming Interface) and to dispatch them to the appropriate driver for actual execution. And finally, DBI is responsible of taking results from the driver and giving back it to the calling script.

### Notation and Conventions

Throughout this chapter following notations will be used and it is recommended that you should also follow the same convention.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$dsn</td>
<td>Database source name</td>
</tr>
<tr>
<td>$dbh</td>
<td>Database handle object</td>
</tr>
<tr>
<td>$sth</td>
<td>Statement handle object</td>
</tr>
<tr>
<td>$h</td>
<td>Any of the handle types above ($dbh, $sth, or $drh)</td>
</tr>
</tbody>
</table>
Database Connection

Assuming we are going to work with MySQL database. Before connecting to a database make sure of the followings. You can take help of our MySQL tutorial in case you are not aware about how to create database and tables in MySQL database.

- You have created a database with a name TESTDB.
- You have created a table with a name TEST_TABLE in TESTDB.
- This table is having fields FIRST_NAME, LAST_NAME, AGE, SEX and INCOME.
- User ID "testuser" and password "test123" are set to access TESTDB.
- Perl Module DBI is installed properly on your machine.
- You have gone through MySQL tutorial to understand MySQL Basics.

Following is the example of connecting with MySQL database "TESTDB":

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

use DBI;
use strict;

my $driver = "mysql";
my $database = "TESTDB";
my $dsn = "DBI:$driver:database=$database";
my $userid = "testuser";
my $password = "test123";
```
my $dbh = DBI->connect($dsn, $userid, $password ) or die $DBI::errstr;

If a connection is established with the datasource then a Database Handle is returned and saved into $dbh for further use otherwise $dbh is set to undef value and $DBI::errstr returns an error string.

**INSERT Operation**

INSERT operation is required when you want to create some records into a table. Here we are using table TEST_TABLE to create our records. So once our database connection is established, we are ready to create records into TEST_TABLE. Following is the procedure to create single record into TEST_TABLE. You can create as many as records you like using the same concept.

Record creation takes the following steps:

- Preparing SQL statement with INSERT statement. This will be done using prepare() API.
- Executing SQL query to select all the results from the database. This will be done using execute() API.
- Releasing Statement handle. This will be done using finish() API.
- If everything goes fine then commit this operation otherwise you can rollback complete transaction. Commit and Rollback are explained in next sections.

```perl
my $sth = $dbh->prepare("INSERT INTO TEST_TABLE
                         (FIRST_NAME, LAST_NAME, SEX, AGE, INCOME )
                         values
                         ('john', 'poul', 'M', 30, 13000)");
$sth->execute() or die $DBI::errstr;
$sth->finish();
$dbh->commit or die $DBI::errstr;
```

**Using Bind Values**

There may be a case when values to be entered is not given in advance. So you can use bind variables which will take the required values at run time. Perl DBI modules make use of a question mark in place of actual value and then actual values are passed through execute() API at the run time. Following is the example:

```perl
my $dbh = DBI->connect($dsn, $userid, $password ) or die $DBI::errstr;
```
my $first_name = "john";
my $last_name = "poul";
my $sex = "M";
my $income = 13000;
my $age = 30;
my $sth = $dbh->prepare("INSERT INTO TEST_TABLE
    (FIRST_NAME, LAST_NAME, SEX, AGE, INCOME )
    values
    (?,?,?,?,?)");
$sth->execute($first_name,$last_name,$sex, $age, $income)
    or die $DBI::errstr;
$sth->finish();
$dbh->commit or die $DBI::errstr;

READ Operation

READ Operation on any database means to fetch some useful information from
the database, i.e., one or more records from one or more tables. So once our
database connection is established, we are ready to make a query into this
database. Following is the procedure to query all the records having AGE greater
than 20. This will take four steps:

- Preparing SQL SELECT query based on required conditions. This will be
done using prepare() API.
- Executing SQL query to select all the results from the database. This will
be done using execute() API.
- Fetching all the results one by one and printing those results. This will be
done using fetchrow_array() API.
- Releasing Statement handle. This will be done using finish() API.

my $sth = $dbh->prepare("SELECT FIRST_NAME, LAST_NAME
    FROM TEST_TABLE
    WHERE AGE > 20");
$sth->execute() or die $DBI::errstr;
print "Number of rows found :" + $sth->rows;
while (my @row = $sth->fetchrow_array()) {
    my ($first_name, $last_name ) = @row;
print "First Name = $first_name, Last Name = $last_name\n";
}
$sth->finish;

Using Bind Values

There may be a case when condition is not given in advance. So you can use bind variables, which will take the required values at run time. Perl DBI modules makes use of a question mark in place of actual value and then the actual values are passed through execute() API at the run time. Following is the example:

$age = 20;
my $sth = $dbh->prepare("SELECT FIRST_NAME, LAST_NAME
               FROM TEST_TABLE
               WHERE AGE > ?");
$sth->execute( $age ) or die $DBI::errstr;
print "Number of rows found :" + $sth->rows;
while (my @row = $sth->fetchrow_array()) {
    my ($first_name, $last_name ) = @row;
    print "First Name = $first_name, Last Name = $last_name\n";
}
$sth->finish();

UPDATE Operation

UPDATE Operation on any database means to update one or more records already available in the database tables. Following is the procedure to update all the records having SEX as 'M'. Here we will increase AGE of all the males by one year. This will take three steps:

- Preparing SQL query based on required conditions. This will be done using prepare() API.
- Executing SQL query to select all the results from the database. This will be done using execute() API.
- Releasing Statement handle. This will be done using finish() API.
- If everything goes fine then commit this operation otherwise you can rollback complete transaction. See next section for commit and rollback APIs.
my $sth = $dbh->prepare("UPDATE TEST_TABLE
    SET   AGE = AGE + 1
    WHERE SEX = 'M'");

$sth->execute() or die $DBI::errstr;

print "Number of rows updated :" + $sth->rows;

$sth->finish();

$dbh->commit or die $DBI::errstr;

**Using Bind Values**

There may be a case when condition is not given in advance. So you can use bind variables, which will take required values at run time. Perl DBI modules make use of a question mark in place of actual value and then the actual values are passed through execute() API at the run time. Following is the example:

```perl
$sex = 'M';
my $sth = $dbh->prepare("UPDATE TEST_TABLE
    SET   AGE = AGE + 1
    WHERE SEX = ?");

$sth->execute('$_sex') or die $DBI::errstr;

print "Number of rows updated :" + $sth->rows;

$sth->finish();

$dbh->commit or die $DBI::errstr;
```

In some case you would like to set a value, which is not given in advance so you can use binding value as follows. In this example income of all males will be set to 10000.

```perl
$sex = 'M';

$income = 10000;

my $sth = $dbh->prepare("UPDATE TEST_TABLE
    SET   INCOME = ?
    WHERE SEX = ?");

$sth->execute( $income, '$_sex') or die $DBI::errstr;

print "Number of rows updated :" + $sth->rows;

$sth->finish();
```
DELETE Operation

DELETE operation is required when you want to delete some records from your database. Following is the procedure to delete all the records from TEST_TABLE where AGE is equal to 30. This operation will take the following steps.

- Preparing SQL query based on required conditions. This will be done using prepare() API.
- Executing SQL query to delete required records from the database. This will be done using execute() API.
- Releasing Statement handle. This will be done using finish() API.
- If everything goes fine then commit this operation otherwise you can rollback complete transaction.

```perl
$age = 30;
my $sth = $dbh->prepare("DELETE FROM TEST_TABLE
    WHERE AGE = ?");
$sth->execute($age) or die $DBI::errstr;
print "Number of rows deleted :" + $sth->rows;
$sth->finish();
$dbh->commit or die $DBI::errstr;
```

Using do Statement

If you're doing an UPDATE, INSERT, or DELETE there is no data that comes back from the database, so there is a short cut to perform this operation. You can use do statement to execute any of the command as follows.

```perl
$dbh->do('DELETE FROM TEST_TABLE WHERE age =30');
```

do returns a true value if it succeeded, and a false value if it failed. Actually, if it succeeds it returns the number of affected rows. In the example it would return the number of rows that were actually deleted.

COMMIT Operation

Commit is the operation which gives a green signal to database to finalize the changes and after this operation no change can be reverted to its original position.

Here is a simple example to call commit API.
ROLLBACK Operation

If you are not satisfied with all the changes or you encounter an error in between of any operation, you can revert those changes to use `rollback` API. Here is a simple example to call `rollback` API.

```perl
$dbh->rollback or die $dbh->errstr;
```

Begin Transaction

Many databases support transactions. This means that you can make a whole bunch of queries which would modify the databases, but none of the changes are actually made. Then at the end, you issue the special SQL query `COMMIT`, and all the changes are made simultaneously. Alternatively, you can issue the query `ROLLBACK`, in which case all the changes are thrown away and database remains unchanged.

Perl DBI module provided `begin_work` API, which enables transactions (by turning AutoCommit off) until the next call to commit or rollback. After the next commit or rollback, AutoCommit will automatically be turned on again.

```perl
$rc = $dbh->begin_work or die $dbh->errstr;
```

AutoCommit Option

If your transactions are simple, you can save yourself the trouble of having to issue a lot of commits. When you make the connect call, you can specify an `AutoCommit` option which will perform an automatic commit operation after every successful query. Here's what it looks like:

```perl
my $dbh = DBI->connect($dsn, $userid, $password,
   {AutoCommit => 1})
   or die $DBI::errstr;
```

Here AutoCommit can take value 1 or 0, where 1 means AutoCommit is on and 0 means AutoCommit is off.

Automatic Error Handling

When you make the connect call, you can specify a `RaiseErrors` option that handles errors for you automatically. When an error occurs, DBI will abort your execution.
program instead of returning a failure code. If all you want is to abort the program on an error, this can be convenient. Here’s what it looks like:

```perl
my $dbh = DBI->connect($dsn, $userid, $password,
    {RaiseError => 1});
    or die $DBI::errstr;
```

Here RaiseError can take value 1 or 0.

**Disconnecting Database**

To disconnect Database connection, use `disconnect` API as follows:

```perl
$rc = $dbh->disconnect or warn $dbh->errstr;
```

The transaction behaviour of the disconnect method is, sadly, undefined. Some database systems (such as Oracle and Ingres) will automatically commit any outstanding changes, but others (such as Informix) will rollback any outstanding changes. Applications not using `AutoCommit` should explicitly call commit or rollback before calling disconnect.

**Using NULL Values**

Undefined values, or `undef`, are used to indicate NULL values. You can insert and update columns with a NULL value as you would a non-NULL value. These examples insert and update the column age with a NULL value:

```perl
$sth = $dbh->prepare(qq{
    INSERT INTO TEST_TABLE (FIRST_NAME, AGE) VALUES (?, ?)
});
$sth->execute("Joe", undef);
```

Here `qq{}` is used to return a quoted string to `prepare` API. However, care must be taken when trying to use NULL values in a WHERE clause. Consider:

```sql
SELECT FIRST_NAME FROM TEST_TABLE WHERE age = ?
```

Binding an `undef` (NULL) to the placeholder will not select rows, which have a NULL age! At least for database engines that conform to the SQL standard. Refer to the SQL manual for your database engine or any SQL book for the reasons for this. To explicitly select NULLs you have to say "WHERE age IS NULL".

A common issue is to have a code fragment handle a value that could be either defined or `undef` (non-NULL or NULL) at runtime. A simple technique is to
prepare the appropriate statement as needed, and substitute the placeholder for non-NULL cases:

```perl
$sql_clause = defined $age ? "age = ?" : "age IS NULL";
$sth = $dbh->prepare(qq{
   SELECT FIRST_NAME FROM TEST_TABLE WHERE $sql_clause
});
$sth->execute(defined $age ? $age : ());
```

### Some Other DBI Functions

**available_drivers**

```perl
@ary = DBI->available_drivers;
@ary = DBI->available_drivers($quiet);
```

Returns a list of all available drivers by searching for DBD::* modules through the directories in @INC. By default, a warning is given if some drivers are hidden by others of the same name in earlier directories. Passing a true value for $quiet will inhibit the warning.

**installed_drivers**

```perl
%drivers = DBI->installed_drivers();
```

Returns a list of driver name and driver handle pairs for all drivers 'installed' (loaded) into the current process. The driver name does not include the 'DBD::' prefix.

**data_sources**

```perl
@ary = DBI->data_sources($driver);
```

Returns a list of data sources (databases) available via the named driver. If $driver is empty or undef, then the value of the DBI_DRIVER environment variable is used.

**quote**

```perl
$sql = $dbh->quote($value);
$sql = $dbh->quote($value, $data_type);
```
Quote a string literal for use as a literal value in an SQL statement, by escaping any special characters (such as quotation marks) contained within the string and adding the required type of outer quotation marks.

```perl
$sql = sprintf "SELECT foo FROM bar WHERE baz = %s",
        $dbh->quote("Don't");
```

For most database types, quote would return 'Don"t' (including the outer quotation marks). It is valid for the quote() method to return an SQL expression that evaluates to the desired string. For example:

```perl
$quoted = $dbh->quote("one\ntwo\0three")
```

may produce results which will be equivalent to

```sql
CONCAT('one', CHAR(12), 'two', CHAR(0), 'three')
```

## Methods Common to All Handles

### err

```perl
$rv = $h->err;
```

or

```perl
$rv = $DBI::err
```

or

```perl
$rv = $h->err
```

Returns the native database engine error code from the last driver method called. The code is typically an integer but you should not assume that. This is equivalent to $DBI::err or $h->err.

### errstr

```perl
$str = $h->errstr;
```

or

```perl
$str = $DBI::errstr
```

or

```perl
$str = $h->errstr
```

Returns a string representation of the error code.
Returns the native database engine error message from the last DBI method called. This has the same lifespan issues as the "err" method described above. This is equivalent to $DBI::errstr or $h->errstr.

**rows**

```perl
$rv = $h->rows;
```

or

```perl
$rv = $DBI::rows
```

This returns the number of rows effected by previous SQL statement and equivalent to $DBI::rows.

**trace**

```perl
$h->trace($trace_settings);
```

DBI sports an extremely useful ability to generate runtime tracing information of what it's doing, which can be a huge time-saver when trying to track down strange problems in your DBI programs. You can use different values to set trace level. These values varies from 0 to 4. The value 0 means disable trace and 4 means generate complete trace.

**Interpolated Statements are Prohibited**

It is highly recommended not to use interpolated statements as follows:

```perl
while ($first_name = <>) {
    my $sth = $dbh->prepare("SELECT *
                            FROM TEST_TABLE
                            WHERE FIRST_NAME = '$first_name'");

    $sth->execute();
    # and so on ...
}
```

Thus don't use interpolated statement instead use bind value to prepare dynamic SQL statement.
What is CGI?

- A Common Gateway Interface, or CGI, is a set of standards that defines how information is exchanged between the web server and a custom script.
- The CGI specs are currently maintained by the NCSA and NCSA defines CGI as follows:
  - The Common Gateway Interface, or CGI, is a standard for external gateway programs to interface with information servers such as HTTP servers.
- The current version is CGI/1.1 and CGI/1.2 is under progress.

Web Browsing

To understand the concept of CGI, let's see what happens when we click a hyperlink available on a web page to browse a particular web page or URL.

- Your browser contacts web server using HTTP protocol and demands for the URL, i.e., web page filename.
- Web Server will check the URL and will look for the filename requested. If web server finds that file then it sends the file back to the browser without any further execution otherwise sends an error message indicating that you have requested a wrong file.
- Web browser takes response from web server and displays either the received file content or an error message in case file is not found.

However, it is possible to set up HTTP server in such a way so that whenever a file in a certain directory is requested that file is not sent back; instead it is executed as a program, and whatever that program outputs as a result, that is sent back for your browser to display. This can be done by using a special functionality available in the web server and it is called **Common Gateway Interface** or CGI and such programs which are executed by the server to produce final result, are called CGI scripts. These CGI programs can be a PERL Script, Shell Script, C or C++ program, etc.
Web Server Support and Configuration

Before you proceed with CGI Programming, make sure that your Web Server supports CGI functionality and it is configured to handle CGI programs. All the CGI programs to be executed by the web server are kept in a pre-configured directory. This directory is called CGI directory and by convention it is named as /cgi-bin. By convention Perl CGI files will have extension as .cgi.

First CGI Program

Here is a simple link which is linked to a CGI script called hello.cgi. This file has been kept in /cgi-bin/ directory and it has the following content. Before running your CGI program, make sure you have change mode of file using chmod 755 hello.cgi UNIX command.

```
#!/usr/bin/perl
```
```
print "Content-type:text/html\r\n\r\n";
print '<html>';
print '<head>';
print '<title>Hello Word - First CGI Program</title>';
print '</head>';
print '<body>';
print '<h2>Hello Word! This is my first CGI program</h2>';
print '</body>';
print '</html>';
1;
```

Now if you click **hello.cgi** link then request goes to web server who search for hello.cgi in /cgi-bin directory, execute it and whatever result got generated, web server sends that result back to the web browser, which is as follows:

```
Hello Word! This is my first CGI program
```

This hello.cgi script is a simple Perl script which is writing its output on STDOUT file, i.e., screen. There is one important and extra feature available which is first line to be printed **Content-type:text/html\r\n\r\n**. This line is sent back to the browser and specifies the content type to be displayed on the browser screen. Now you must have understood basic concept of CGI and you can write many complicated CGI programs using Perl. This script can interact with any other external system also to exchange information such as a database, web services, or any other complex interfaces.

**Understanding HTTP Header**

The very first line **Content-type:text/html\r\n\r\n** is a part of HTTP header, which is sent to the browser so that browser can understand the incoming content from server side. All the HTTP header will be in the following form:

```
HTTP Field Name: Field Content
```

For Example:

```
Content-type:text/html\r\n\r\n
```

There are few other important HTTP headers, which you will use frequently in your CGI Programming.
<table>
<thead>
<tr>
<th>Header</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content-type: String</td>
<td>A MIME string defining the format of the content being returned. Example is Content-type:text/html</td>
</tr>
<tr>
<td>Expires: Date String</td>
<td>The date when the information becomes invalid. This should be used by the browser to decide when a page needs to be refreshed. A valid date string should be in the format 01 Jan 1998 12:00:00 GMT.</td>
</tr>
<tr>
<td>Location: URL String</td>
<td>The URL that should be returned instead of the URL requested. You can use this field to redirect a request to any other location.</td>
</tr>
<tr>
<td>Last-modified: String</td>
<td>The date of last modification of the file.</td>
</tr>
<tr>
<td>Content-length: String</td>
<td>The length, in bytes, of the data being returned. The browser uses this value to report the estimated download time for a file.</td>
</tr>
<tr>
<td>Set-Cookie: String</td>
<td>Set the cookie passed through the string</td>
</tr>
</tbody>
</table>

### CGI Environment Variables

All the CGI program will have access to the following environment variables. These variables play an important role while writing any CGI program.

<table>
<thead>
<tr>
<th>Variable Names</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTENT_TYPE</td>
<td>The data type of the content. Used when the client is sending attached content to the server. For example file upload, etc.</td>
</tr>
<tr>
<td>CONTENT_LENGTH</td>
<td>The length of the query information. It's available only for POST requests</td>
</tr>
<tr>
<td>HTTP_COOKIE</td>
<td>Returns the set cookies in the form of key &amp; value</td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HTTP_USER_AGENT</td>
<td>The User-Agent request-header field contains information about the user agent originating the request. Its name of the web browser.</td>
</tr>
<tr>
<td>PATH_INFO</td>
<td>The path for the CGI script.</td>
</tr>
<tr>
<td>QUERY_STRING</td>
<td>The URL-encoded information that is sent with GET method request.</td>
</tr>
<tr>
<td>REMOTE_ADDR</td>
<td>The IP address of the remote host making the request. This can be useful for logging or for authentication purpose.</td>
</tr>
<tr>
<td>REMOTE_HOST</td>
<td>The fully qualified name of the host making the request. If this information is not available then REMOTE_ADDR can be used to get IP address.</td>
</tr>
<tr>
<td>REQUEST_METHOD</td>
<td>The method used to make the request. The most common methods are GET and POST.</td>
</tr>
<tr>
<td>SCRIPT_FILENAME</td>
<td>The full path to the CGI script.</td>
</tr>
<tr>
<td>SCRIPT_NAME</td>
<td>The name of the CGI script.</td>
</tr>
<tr>
<td>SERVER_NAME</td>
<td>The server's hostname or IP Address.</td>
</tr>
<tr>
<td>SERVER_SOFTWARE</td>
<td>The name and version of the software the server is running.</td>
</tr>
</tbody>
</table>

Here is a small CGI program to list down all the CGI variables supported by your Web server.

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

print "Content-type: text/html\n\n"
print "<font size=+1>Environment</font>\n";
foreach (sort keys %ENV)
```
Raise a "File Download" Dialog Box?

Sometime it is desired that you want to give option where a user will click a link and it will pop up a "File Download" dialogue box to the user instead of displaying actual content. This is very easy and will be achieved through HTTP headers.

This HTTP header will be different from the header mentioned in previous section. For example, if you want to make a `FileName` file downloadable from a given link then it's syntax will be as follows:

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

# HTTP Header
print "Content-Type: application/octet-stream; name=""FileName""\n"; 
print "Content-Disposition: attachment; filename=""FileName""\n"; 

# Actual File Content will go hear.
open( FILE, "<FileName" );
while(read(FILE, $buffer, 100) )
{
    print("$buffer" );
}
```

GET and POST Methods

You must have come across many situations when you need to pass some information from your browser to the web server and ultimately to your CGI Program handling your requests. Most frequently browser uses two methods to pass this information to the web server. These methods are **GET Method** and **POST Method**. Let's check them one by one.
Passing Information using GET Method

The GET method sends the encoded user information appended to the page URL itself. The page and the encoded information are separated by the ? character as follows:

```
http://www.test.com/cgi-bin/hello.cgi?key1=value1&key2=value2
```

The GET method is the default method to pass information from a browser to the web server and it produces a long string that appears in your browser's Location: box. You should never use GET method if you have password or other sensitive information to pass to the server. The GET method has size limitation: only 1024 characters can be passed in a request string.

This information is passed using QUERY_STRING header and will be accessible in your CGI Program through QUERY_STRING environment variable which you can parse and use in your CGI program.

You can pass information by simply concatenating key and value pairs along with any URL or you can use HTML <FORM> tags to pass information using GET method.

Simple URL Example : Get Method

Here is a simple URL which will pass two values to hello_get.cgi program using GET method.

```
http://www.tutorialspoint.com/cgi-bin/hello_get.cgi?first_name=ZARA&last_name=ALI
```

Below is hello_get.cgi script to handle input given by web browser.

```
#!/usr/bin/perl

local ($buffer, @pairs, $pair, $name, $value, %FORM);

# Read in text
$ENV{'REQUEST_METHOD'} =~ tr/a-z/A-Z/;
if ($ENV{'REQUEST_METHOD'} eq "GET")
{
    $buffer = $ENV{'QUERY_STRING'};
}

# Split information into name/value pairs
@pairs = split(/&/, $buffer);
foreach $pair (@pairs)
```
Simple FORM Example: GET Method

Here is a simple example, which passes two values using HTML FORM and submit button. We are going to use the same CGI script hello_get.cgi to handle this input.

```perl
{
  ($name, $value) = split(/=/, $pair);
  $value =~ tr/+//;
  $value =~ s/%(.)/pack("C", hex($1))/eg;
  $FORM{$name} = $value;
}
$first_name = $FORM{first_name};
$last_name  = $FORM{last_name};

print "Content-type:text/html\r\n\r\n";
print "<html>";
print "<head>";
print "<title>Hello - Second CGI Program</title>";
print "</head>";
print "<body>";
print "<h2>Hello $first_name $last_name - Second CGI Program</h2>";
print "</body>";
print "</html>";
1;
```

Here is the actual output of the above form coding. Now you can enter First and Last Name and then click submit button to see the result.
Passing Information using POST Method

A more reliable method of passing information to a CGI program is the **POST** method. This packages the information in exactly the same way as GET methods, but instead of sending it as a text string after a `?` in the URL, it sends it as a separate message as a part of HTTP header. Web server provides this message to the CGI script in the form of the standard input.

Below is the modified **hello_post.cgi** script to handle input given by the web browser. This script will handle GET as well as POST method.

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

local ($buffer, @pairs, $pair, $name, $value, %FORM);

# Read in text
$ENV{'REQUEST_METHOD'} =~ tr/a-z/A-Z/;
if ($ENV{'REQUEST_METHOD'} eq "POST") {
    read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
} else {
    $buffer = $ENV{'QUERY_STRING'};
}

# Split information into name/value pairs
@pairs = split(/&/, $buffer);
foreach $pair (@pairs) {
    ($name, $value) = split(/=/, $pair);
    $value =~ tr/+//;
    $value =~ s/%(..)/pack("C", hex($1))/eg;
    $FORM{$name} = $value;
}

$first_name = $FORM{first_name};
$last_name = $FORM{last_name};
```
print "Content-type:text/html\r\n\r\n";
print "<html>";
print "<head>";
print "<title>Hello - Second CGI Program</title>";
print "</head>";
print "<body>";
print "<h2>Hello $first_name $last_name - Second CGI Program</h2>";
print "</body>";
print "</html>";

1;

Let us take again same examle as above, which passes two values using HTML FORM and submit button. We are going to use CGI script hello_post.cgi to handle this input.

```html
<form action="/cgi-bin/hello_post.cgi" method="POST">
First Name: <input type="text" name="first_name" > <br>

Last Name: <input type="text" name="last_name" ></input>

<input type="submit" value="Submit">
</form>
```

Here is the actual output of the above form coding, You enter First and Last Name and then click submit button to see the result.

First Name: 

Last Name: 

Passing Checkbox Data to CGI Program

Checkboxes are used when more than one option is required to be selected. Here is an example HTML code for a form with two checkboxes.

```html
<form action="/cgi-bin/checkbox.cgi" method="POST" target="_blank">
```

![tutorialspoint logo](image)
The result of this code is the following form:

- Maths
- Physics

Below is **checkbox.cgi** script to handle input given by web browser for radio button.

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

local ($buffer, @pairs, $pair, $name, $value, %FORM);
# Read in text
$ENV{'REQUEST_METHOD'} =~ tr/a-z/A-Z/;
if ($ENV{'REQUEST_METHOD'} eq "POST") {
    read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
} else {
    $buffer = $ENV{'QUERY_STRING'};
}
# Split information into name/value pairs
@pairs = split(/&/, $buffer);
foreach $pair (@pairs) {
    ($name, $value) = split(/=/, $pair);
    $value =~ tr/+//;
    $value =~ s/%(.)/pack("C", hex($1))/eg;
    $FORM{$name} = $value;
}
if( $FORM{maths} ){
    $maths_flag = "ON";
} else{
    $maths_flag = "OFF";

```
if( $FORM{physics} ){
    $physics_flag = "ON";
}else{
    $physics_flag = "OFF";
}

print "Content-type:text/html\n\n\n";
print "<html>";
print "<head>";
print "<title>Checkbox - Third CGI Program</title>";
print "</head>";
print "<body>";
print "<h2> CheckBox Maths is : $maths_flag</h2>";
print "<h2> CheckBox Physics is : $physics_flag</h2>";
print "</body>";
print "</html>";
1;

Passing Radio Button Data to CGI Program

Radio Buttons are used when only one option is required to be selected. Here is an example HTML code for a form with two radio button:

<form action="/cgi-bin/radiobutton.cgi" method="POST" target="_blank">
<input type="radio" name="subject" value="maths"> Maths
<input type="radio" name="subject" value="physics"> Physics
<input type="submit" value="Select Subject">
</form>

The result of this code is the following form:

☐ Maths ☐ Physics Submit

Below is radiobutton.cgi script to handle input given by the web browser for radio button.
#!/usr/bin/perl

local ($buffer, @pairs, $pair, $name, $value, %FORM);

# Read in text
$ENV{'REQUEST_METHOD'} =~ tr/a-z/A-Z/;
if ($ENV{'REQUEST_METHOD'} eq "POST")
{
    read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
} else {
    $buffer = $ENV{'QUERY_STRING'};
}

# Split information into name/value pairs
@pairs = split(/&/, $buffer);
foreach $pair (@pairs)
{
    ($name, $value) = split(/=/, $pair);
    $value =~ tr/+/*/;
    $value =~ s/%(.)/pack("C", hex($1))/eg;
    $FORM{$name} = $value;
}

$subject = $FORM{subject};

print "Content-type:text/html\n\n\n\n";
print "<html>";
print "<head>";
print "<title>Radio - Fourth CGI Program</title>";
print "</head>";
print "<body>";
print "<h2> Selected Subject is $subject</h2>";
print "</body>";
print "</html>";

1;
Passing Text Area Data to CGI Program

A textarea element is used when multiline text has to be passed to the CGI Program. Here is an example HTML code for a form with a TEXTAREA box:

```html
<form action="/cgi-bin/textarea.cgi" method="POST" target="_blank">
  <textarea name="textcontent" cols=40 rows=4>
Type your text here...
  </textarea>
  <input type="submit" value="Submit">
</form>
```

The result of this code is the following form:

![Textarea form](Image)

Below is the `textarea.cgi` script to handle input given by the web browser.

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

local ($buffer, @pairs, $pair, $name, $value, %FORM);
# Read in text
$ENV{ 'REQUEST_METHOD' } =~ tr/a-z/A-Z/;
if ($ENV{'REQUEST_METHOD'} eq "POST") {
  read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
} else {
  $buffer = $ENV{'QUERY_STRING'};
}
# Split information into name/value pairs
@pairs = split(/&/, $buffer);
foreach $pair (@pairs) {
  ($name, $value) = split(/=/, $pair);
  $value =~ tr/+/ /;
  $value =~ s/%(..)/pack("C", hex($1))/eg;
```

---

238
Perl

```perl
$FORM{$name} = $value;
}
$text_content = $FORM{textcontent};

print "Content-type:text/html\n\r\n\r
";
print "<html>";
print "<head>";
print "<title>Text Area - Fifth CGI Program</title>";
print "</head>";
print "<body>";
print "<h2> Entered Text Content is $text_content</h2>";
print "</body>";
print "</html>";
1;
```

**Passing Drop Down Box Data to CGI Program**

A drop down box is used when we have many options available but only one or two will be selected. Here is example HTML code for a form with one drop down box

```
<form action="/cgi-bin/dropdown.cgi" method="POST" target="_blank">
<select name="dropdown">
<option value="Maths" selected>Maths</option>
<option value="Physics">Physics</option>
</select>
<input type="submit" value="Submit">
</form>
```

The result of this code is the following form:

![Drop Down Box Example](image)

Below is the **dropdown.cgi** script to handle input given by web browser.

```
#!/usr/bin/perl
```
local ($buffer, @pairs, $pair, $name, $value, %FORM);
# Read in text
$ENV{'REQUEST_METHOD'} =~ tr/a-z/A-Z/;
if ($ENV{'REQUEST_METHOD'} eq "POST")
{
    read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
}
else {
    $buffer = $ENV{'QUERY_STRING'};
}
# Split information into name/value pairs
@pairs = split(/&/, $buffer);
foreach $pair (@pairs)
{
    ($name, $value) = split(/=/, $pair);
    $value =~ tr/+//;
    $value =~ s/%(..)/pack("C", hex($1))/eg;
    $FORM{$name} = $value;
}
$subject = $FORM{dropdown};

print "Content-type:text/html\n\n\r\n\n";
print "<html>
";
print "<head>
";
print "<title>Dropdown Box - Sixth CGI Program</title>
";
print "</head>
";
print "<body>
";
print "<h2> Selected Subject is $subject</h2>
";
print "</body>
";
print "</html>
";
1;
Using Cookies in CGI

HTTP protocol is a stateless protocol. But for a commercial website it is required to maintain session information among different pages. For example one user registration ends after transactions which spans through many pages. But how to maintain user’s session information across all the web pages?

In many situations, using cookies is the most efficient method of remembering and tracking preferences, purchases, commissions, and other information required for better visitor experience or site statistics.

How It Works

Your server sends some data to the visitor’s browser in the form of a cookie. The browser may accept the cookie. If it does, it is stored as a plain text record on the visitor’s hard drive. Now, when the visitor arrives at another page on your site, the cookie is available for retrieval. Once retrieved, your server knows/remembers what was stored.

Cookies are a plain text data record of 5 variable-length fields:

- **Expires**: The date the cookie will expire. If this is blank, the cookie will expire when the visitor quits the browser.
- **Domain**: The domain name of your site.
- **Path**: The path to the directory or web page that set the cookie. This may be blank if you want to retrieve the cookie from any directory or page.
- **Secure**: If this field contains the word "secure" then the cookie may only be retrieved with a secure server. If this field is blank, no such restriction exists.
- **Name=Value**: Cookies are set and retrieved in the form of key and value pairs.

Setting up Cookies

It is very easy to send cookies to browser. These cookies will be sent along with the HTTP Header. Assuming you want to set UserID and Password as cookies. So it will be done as follows:

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

print "Set-Cookie:UserID=XYZ;\n";
print "Set-Cookie:Password=XYZ123;\n";
print "Set-Cookie:Expires=Tuesday, 31-Dec-2007 23:12:40 GMT";\n";
```
Here we used **Set-Cookie** HTTP header to set cookies. It is optional to set cookies attributes like Expires, Domain, and Path. It is important to note that cookies are set before sending magic line "**Content-type:text/html\r\n\r\n**."

### Retrieving Cookies

It is very easy to retrieve all the set cookies. Cookies are stored in CGI environment variable HTTP_COOKIE and they will have following form.

```
key1=value1;key2=value2;key3=value3....
```

Here is an example of how to retrieve cookies.

```perl
#!/usr/bin/perl
$rcvd_cookies = $ENV{'HTTP_COOKIE'};
@cookies = split /;/, $rcvd_cookies;
foreach $cookie (@cookies){
    ($key, $val) = split(/=/, $cookie); # splits on the first =.
    $key =~ s/^\s+//;
    $val =~ s/^\s+//;
    $key =~ s/$\s+//;
    $val =~ s/$\s+//;
    if( $key eq "UserID" ){
        $user_id = $val;
    }elsif($key eq "Password"){
        $password = $val;
    }
}
print "User ID = $user_id\n";
print "Password = $password\n";
```

This will produce the following result, provided above cookies have been set before calling retrieval cookies script.
User ID = XYZ
Password = XYZ123

**CGI Modules and Libraries**

You will find many built-in modules over the internet which provides you direct functions to use in your CGI program. Following are the important ones.

- **CGI Module**
- **Berkeley cgi-lib.pl**
What are Packages?

The `package` statement switches the current naming context to a specified namespace (symbol table). Thus:

- A package is a collection of code which lives in its own namespace.
- A namespace is a named collection of unique variable names (also called a symbol table).
- Namespaces prevent variable name collisions between packages.
- Packages enable the construction of modules which, when used, won't clobber variables and functions outside of the module's own namespace.
- The package stays in effect until either another package statement is invoked, or until the end of the current block or file.
- You can explicitly refer to variables within a package using the `::` package qualifier.

Following is an example having main and Foo packages in a file. Here special variable `__PACKAGE__` has been used to print the package name.

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

# This is main package
$i = 1;
print "Package name : " , __PACKAGE__ , " $i\n";

package Foo;
# This is Foo package
$i = 10;
print "Package name : " , __PACKAGE__ , " $i\n";

package main;
# This is again main package
$i = 100;
```
print "Package name : " , __PACKAGE__ , " $i\n";
print "Package name : " , __PACKAGE__ , " $Foo::i\n";

1;

When above code is executed, it produces the following result:

Package name : main 1
Package name : Foo 10
Package name : main 100
Package name : main 10

BEGIN and END Blocks

You may define any number of code blocks named BEGIN and END, which act as constructors and destructors respectively.

BEGIN { ... }
END { ... }
BEGIN { ... }
END { ... }

- Every BEGIN block is executed after the perl script is loaded and compiled but before any other statement is executed.
- Every END block is executed just before the perl interpreter exits.
- The BEGIN and END blocks are particularly useful when creating Perl modules.

Following example shows its usage:

#!/usr/bin/perl

package Foo;
print "Begin and Block Demo\n";

BEGIN {
    print "This is BEGIN Block\n"
}

1;
END {
    print "This is END Block\n"
}

1;

When above code is executed, it produces the following result:

This is BEGIN Block
Begin and Block Demo
This is END Block

What are Perl Modules?

A Perl module is a reusable package defined in a library file whose name is the
same as the name of the package with a .pm as extension.

A Perl module file called Foo.pm might contain statements like this.

#!/usr/bin/perl

package Foo;
sub bar {
    print "Hello $_[0]\n"
}

sub blat {
    print "World $_[0]\n"
}

1;

Few important points about Perl modules

- The functions require and use will load a module.
- Both use the list of search paths in @INC to find the module.
- Both functions require and use call the eval function to process the code.
- The 1; at the bottom causes eval to evaluate to TRUE (and thus not fail).
The Require Function

A module can be loaded by calling the `require` function as follows:

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

require Foo;

Foo::bar( "a" );
Foo::blat( "b" );
```

You must have noticed that the subroutine names must be fully qualified to call them. It would be nice to enable the subroutine `bar` and `blat` to be imported into our own namespace so we wouldn't have to use the `Foo::` qualifier.

The Use Function

A module can be loaded by calling the `use` function.

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

use Foo;

bar( "a" );
blat( "b" );
```

Notice that we didn't have to fully qualify the package's function names. The `use` function will export a list of symbols from a module given a few added statements inside a module.

```perl
require Exporter;
@ISA = qw(Exporter);
```

Then, provide a list of symbols (scalars, lists, hashes, subroutines, etc) by filling the list variable named `@EXPORT`: For Example:

```perl
package Module;

require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(bar blat);
```
sub bar { print "Hello $_[0]\n" }  
sub blat { print "World $_[0]\n" }  
sub splat { print "Not $_[0]\n" } # Not exported!  
1;

Create the Perl Module Tree

When you are ready to ship your Perl module, then there is standard way of creating a Perl Module Tree. This is done using `h2xs` utility. This utility comes along with Perl. Here is the syntax to use h2xs:

```
$h2xs -AX -n ModuleName
```

For example, if your module is available in `Person.pm` file, then simply issue the following command:

```
$h2xs -AX -n Person
```

This will produce the following result:

```
Writing Person/lib/Person.pm  
Writing Person/Makefile.PL  
Writing Person/README  
Writing Person/t/Person.t  
Writing Person/Changes  
Writing Person/MANIFEST
```

Here is the description of these options:

- `-A` omits the Autoloader code (best used by modules that define a large number of infrequently used subroutines).
- `-X` omits XS elements (eXternal Subroutine, where eXternal means external to Perl, i.e., C).
- `-n` specifies the name of the module.

So above command creates the following structure inside Person directory. Actual result is shown above.

- Changes
- Makefile.PL
- MANIFEST (contains the list of all files in the package)
- README
- t/ (test files)
- lib/ (Actual source code goes here)

So finally, you `tar` this directory structure into a file Person.tar.gz and you can ship it. You will have to update README file with the proper instructions. You can also provide some test examples files in t directory.

### Installing Perl Module

Download a Perl module in the form tar.gz file. Use the following sequence to install any Perl Module `Person.pm` which has been downloaded in as `Person.tar.gz` file.

```
tar xvfz Person.tar.gz
cd Person
perl Makefile.PL
make
make install
```

The Perl interpreter has a list of directories in which it searches for modules (global array @INC).
You can use Perl in various ways to create new processes as per your requirements. This tutorial will list down few important and most frequently used methods of creating and managing Perl processes.

- You can use special variables $$ or $PROCESS_ID to get current process ID.
- Every process created using any of the mentioned methods, maintains its own virtual environment with-in %ENV variable.
- The exit() function always exits just the child process which executes this function and the main process as a whole will not exit unless all running child-processes have exited.
- All open handles are dup()-ed in child-processes, so that closing any handles in one process does not affect the others.

**Backstick Operator**

This simplest way of executing any Unix command is by using backstick operator. You simply put your command inside the backstick operator, which will result in execution of the command and returns its result which can be stored as follows:

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

@files = `ls -l`;  # The above line executes the 'ls -l' command

foreach $file (@files){  # Loop through the files
    print $file;
}

1;
```

When the above code is executed, it lists down all the files and directories available in the current directory:

```
drwxr-xr-x 3 root root 4096 Sep 14 06:46 9-14
drwxr-xr-x 4 root root 4096 Sep 13 07:54 android
```
The system() Function

You can also use `system()` function to execute any Unix command, whose output will go to the output of the perl script. By default, it is the screen, i.e., STDOUT, but you can redirect it to any file by using redirection operator `>:`

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

system( "ls -l"
)
```

When above code is executed, it lists down all the files and directories available in the current directory:

```
drwxr-xr-x 3 root root 4096 Sep 14 06:46 9-14
drwxr-xr-x 4 root root 4096 Sep 13 07:54 android
-rw-r--r-- 1 root root 574 Sep 17 15:16 index.htm
drwxr-xr-x 3 544 401 4096 Jul 6 16:49 MIME-Lite-3.01
-rw-r--r-- 1 root root 71 Sep 17 15:16 test.pl
drwx------ 2 root root 4096 Sep 17 15:11 vAtrJdy
```

Be careful when your command contains shell environmental variables like $PATH or $HOME. Try following three scenarios:

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

$PATH = "I am Perl Variable";

system('echo $PATH');  # Treats $PATH as shell variable
system("echo $PATH");  # Treats $PATH as Perl variable
system("echo \$PATH");  # Escaping $ works.
```
When above code is executed, it produces the following result depending on what is set in shell variable $PATH.

```
/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin
I am Perl Variable
/usr/local/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin
```

### The fork() Function

Perl provides a `fork()` function that corresponds to the Unix system call of the same name. On most Unix-like platforms where the `fork()` system call is available, Perl's `fork()` simply calls it. On some platforms such as Windows where the `fork()` system call is not available, Perl can be built to emulate `fork()` at the interpreter level.

The `fork()` function is used to clone a current process. This call creates a new process running the same program at the same point. It returns the child pid to the parent process, 0 to the child process, or undef if the fork is unsuccessful.

You can use `exec()` function within a process to launch the requested executable, which will be executed in a separate process area and `exec()` will wait for it to complete before exiting with the same exit status as that process.

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

if(!defined($pid = fork())) {
    # fork returned undef, so unsuccessful
    die "Cannot fork a child: $!";
}elsif ($pid == 0) {
    print "Printed by child process\n";
    exec("date") || die "can't exec date: $!";
}

} else {
    # fork returned 0 nor undef
    # so this branch is parent
    print "Printed by parent process\n";
    $ret = waitpid($pid, 0);
    print "Completed process id: $ret\n";
```
When above code is executed, it produces the following result:

Printed by parent process
Printed by child process
Tue Sep 17 15:41:08 CDT 2013
Completed process id: 17777

The `wait()` and `waitpid()` can be passed as a pseudo-process ID returned by `fork()`. These calls will properly wait for the termination of the pseudo-process and return its status. If you fork without ever waiting on your children using `waitpid()` function, you will accumulate zombies. On Unix systems, you can avoid this by setting `$SIG{CHLD}` to "IGNORE" as follows:

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

local $SIG{CHLD} = "IGNORE";

if (!$defined($pid = fork())) {
    # fork returned undef, so unsuccessful
    die "Cannot fork a child: $!";
} elsif ($pid == 0) {
    print "Printed by child process\n";
    exec("date") || die "can't exec date: $!";
} else {
    # fork returned 0 nor undef
    # so this branch is parent
    print "Printed by parent process\n";
    $ret = waitpid($pid, 0);
    print "Completed process id: $ret\n";
}
```
When above code is executed, it produces the following result:

Printed by parent process
Printed by child process
Tue Sep 17 15:44:07 CDT 2013
Completed process id: -1

The kill() Function

Perl `kill('KILL', (Process List))` function can be used to terminate a pseudo-process by passing it the ID returned by `fork()`.

Note that using `kill('KILL', (Process List))` on a pseudo-process() may typically cause memory leaks, because the thread that implements the pseudo-process does not get a chance to clean up its resources.

You can use `kill()` function to send any other signal to target processes, for example following will send SIGINT to a process IDs 104 and 102:

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

kill('INT', 104, 102);

1;
```
You can embed Pod (Plain Old Text) documentation in your Perl modules and scripts. Following is the rule to use embedded documentation in your Perl Code:

Start your documentation with an empty line, a =head1 command at the beginning, and end it with a =cut command and an empty line.

Perl will ignore the Pod text you entered in the code. Following is a simple example of using embedded documentation inside your Perl code:

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

print "Hello, World
";

=head1 Hello, World Example
This example demonstrate very basic syntax of Perl.
=cut

print "Hello, Universe
";
```

When above code is executed, it produces the following result:

```
Hello, World
Hello, Universe
```

If you're going to put your Pod at the end of the file, and you're using an __END__ or __DATA__ cut mark, make sure to put an empty line there before the first Pod command as follows, otherwise without an empty line before the =head1, many translators wouldn't have recognized the =head1 as starting a Pod block.

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

print "Hello, World
";

while(<DATA>){
    print $_
;
}
```
This example demonstrates very basic syntax of Perl.

```perl
print "Hello, Universe\n";
```

When above code is executed, it produces the following result:

```
Hello, Universe
```

Let's take one more example for the same code without reading the DATA part:

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

print "Hello, World\n";

__END__

=head1 Hello, World Example
This example demonstrates very basic syntax of Perl.
print "Hello, Universe\n";
```

When above code is executed, it produces the following result:

```
Hello, World
```

**What is POD?**

Pod is a simple-to-use markup language used for writing documentation for Perl, Perl programs, and Perl modules. There are various translators available for converting Pod to various formats like plain text, HTML, man pages, and more. Pod markup consists of three basic kinds of paragraphs:

- **Ordinary Paragraph**: You can use formatting codes in ordinary paragraphs, for bold, italic, code-style, hyperlinks, and more.
- **Verbatim Paragraph**: Verbatim paragraphs are usually used for presenting a codeblock or other text which does not require any special parsing or formatting, and which shouldn't be wrapped.

- **Command Paragraph**: A command paragraph is used for special treatment of whole chunks of text, usually as headings or parts of lists. All command paragraphs start with `=`, followed by an identifier, followed by arbitrary text that the command can use however it pleases. Currently recognized commands are:

  ```plaintext
  =pod
  =head1 Heading Text
  =head2 Heading Text
  =head3 Heading Text
  =head4 Heading Text
  =over indentlevel
  =item stuff
  =back
  =begin format
  =end format
  =for format text...
  =encoding type
  =cut
  ```

**POD Examples**

Consider the following POD:

```plaintext
=head1 SYNOPSIS
Copyright 2005 [TUTORIALSOPOINT].
=cut
```

You can use `pod2html` utility available on Linux to convert above POD into HTML, so it will produce following result:

**Copyright 2005 [TUTORIALSOPOINT].**

Next, consider the following example:

```plaintext
=head2 An Example List
```
This is a bulleted list.

Here's another item.

Here's some embedded HTML. In this block I can include images, apply styles, or do anything else I can do with HTML. Pod parsers that aren't outputting HTML will completely ignore it.
Here is the list of all the important functions supported by standard Perl.

**abs**

**Description**
This function returns the absolute value of its argument. If pure integer value is passed then it will return it as it is, but if a string is passed then it will return zero. If VALUE is omitted then it uses $_

**Syntax**
Following is the simple syntax for this function:

```
abs VALUE
```

**Return Value**
This function returns the absolute value of its argument.

**Example**
Following is the example code showing its basic usage:

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

$par1 = 10.25;
$par2 = 20;
$par3 = "ABC";

$abs1 = abs($par1);
$abs2 = abs($par2);
$abs3 = abs($par3);

print "Abs value of par1 is $abs1\n" ;
```
print "Abs value of par2 is $abs2\n" ;
print "Abs value of par3 is $abs3\n" ;

When above code is executed, it produces the following result:

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This function accepts an incoming connection on the existing GENERICSOCKET, which should have been created with socket and bound to a local address using bind. The new socket, which will be used for communication with the client will be NEWSOCKET. GENERICSOCKET will remain unchanged.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Following is the simple syntax for this function:</td>
</tr>
</tbody>
</table>

```perl
accept NEWSOCKET, GENERICSOCKET
```

<table>
<thead>
<tr>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>This function returns 0 on failure and Packed address of remote host on success.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Following is the example code showing script to create a server:</td>
</tr>
</tbody>
</table>

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

When above code is executed, it produces the following result:

<table>
<thead>
<tr>
<th>alarm</th>
</tr>
</thead>
<tbody>
<tr>
<td>This function sets the &quot;alarm,&quot; causing the current process to receive a SIGALRM signal in EXPR seconds. If EXPR is omitted, the value of $_ is used instead.</td>
</tr>
</tbody>
</table>
The actual time delay is not precise, since different systems implement the alarm functionality differently. The actual time may be up to a second more or less than the requested value. You can only set one alarm timer at any one time. If a timer is already running and you make a new call to the alarm function, the alarm timer is reset to the new value. A running timer can be reset without setting a new timer by specifying a value of 0.

**Syntax**

Following is the simple syntax for this function:

```
alarm EXPR

alarm
```

**Return Value**

This function returns Integer value ie. number of seconds remaining for previous timer.

**Example**

Following is the example code showing its basic usage:

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

eval {
    local $SIG{ALRM} = sub { die "alarm\n" }; # NB: \n required
    alarm $timeout;
    $nread = sysread SOCKET, $buffer, $size;
    alarm 0;
};
if ($@) {
    die unless $@ eq "alarm\n"; # propagate unexpected errors
    # timed out
} else {
    # didn't
}
```

When above code is executed, it produces the following result:
atan2

Description
This function returns the arctangent of Y/X in the range -π to π.

Syntax
Following is the simple syntax for this function:

```
atan2 Y, X
```

Return Value
This function returns arctangent of Y/X in the range -π to π.

Example
Following is the example code showing its basic usage:

```
#!/usr/bin/perl

$Y = 30;
$X = 60;

$ret_val = atan2 ($Y, $X);

print "atan2 of 30/60 is : $ret_val\n";
```

When above code is executed, it produces the following result:

```
atan2 of 30/60 is : 0.463647609000806
```
bind

Description
This function binds the network ADDRESS to the filehandle identified by SOCKET. The ADDRESS should be a packed address of the appropriate type for the socket being opened.

Syntax
Following is the simple syntax for this function:

```
bind SOCKET, ADDRESS
```

Return Value
This function returns 0 on failure and 1 on success.

Example
Following is the example code showing its basic usage:

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

When above code is executed, it produces the following result:


binmode

Description
This function sets the format for FILEHANDLE to be read from and written to as binary on the operating systems that differentiate between the two. Files that are not in binary have CR LF sequences converted to LF on input, and LF to CR LF on output. This is vital for operating systems that use two characters to separate lines within text files (MS-DOS), but has no effect on operating systems that use single characters (Unix, Mac OS, QNX).

Syntax
Following is the simple syntax for this function:

```
binmode FILEHANDLE
```
Return Value
This function returns undef on failure or invalid FILEHANDLE and 1 on success.

Example
Following is the example code showing its basic usage:

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

When above code is executed, it produces the following result:
```

bless

Description
This function tells the entity referenced by REF that it is now an object in the CLASSNAME package, or the current package if CLASSNAME is omitted. Use of the two-argument form of bless is recommended.

Syntax
Following is the simple syntax for this function:

```perl
bless REF, CLASSNAME

bless REF
```

Return Value
This function returns the reference to an object blessed into CLASSNAME.

Example
Following is the example code showing its basic usage, the object reference is created by blessing a reference to the package's class.:

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

package Person;
sub new
{
```
my $class = shift;
my $self = {
    _firstName => shift,
    _lastName => shift,
    _ssn => shift,
};
# Print all the values just for clarification.
print "First Name is $self->{_firstName}\n";
print "Last Name is $self->{_lastName}\n";
print "SSN is $self->{_ssn}\n";
bless $self, $class;
return $self;
}

When above code is executed, it produces the following result:

caller

Description
This function returns information about the current subroutines caller. In a scalar context, returns the caller's package name or the package name of the caller EXPR steps up.

In a list context, with no arguments specified, caller returns the package name, file name and line within the file for the caller of the current subroutine.

If EXPR is specified, caller returns extended information for the caller EXPR steps up. That is, when called with an argument of 1, it returns the information for the caller (parent) of the current subroutine, with 2 the caller of the caller (grandparent) of the current subroutine, and so on.

Syntax
Following is the simple syntax for this function:

caller EXPR
Return Value
This function returns undef on failure, basic information when called with no arguments and extended information when called with an argument.

Example
Following is the example code showing its basic usage:

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

When above code is executed, it produces the following result:
```

chdir

Description
This function changes the current working directory to EXPR, or to the user’s home directory if none is specified. This function call is equivalent to Unix command `cd EXPR`.

Syntax
Following is the simple syntax for this function:

```
chdir EXPR
```

Return Value
This function returns 0 on failure and 1 on success.

Example
Following is the example code showing its basic usage, assume you are working in `/user/home/tutorialspoint` directory:

```
#!/usr/bin/perl

chdir "/usr/home";
```
# Now you are in /usr/home dir.
chdir;

# Now you are in home directory /user/home/tutorialspoint

When above code is executed, it produces the following result:

### chmod

#### Description
This function changes the mode of the files specified in LIST to the MODE specified. The value of MODE should be in octal. You must check the return value against the number of files that you attempted to change to determine whether the operation failed. This function call is equivalent to Unix Command chmod MODE FILELIST.

#### Syntax
Following is the simple syntax for this function:

```
chmod MODE, LIST
```

#### Return Value
This function returns Integer, number of files successfully changed.

#### Example
Following is the example code showing its basic usage:

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

$cnt = chmod 0755, 'foo', 'bar';
chmod 0755, @executables;

$mode = '0644'; chmod $mode, 'foo';     # !!! sets mode to ---w----r-T
$mode = '0644'; chmod oct($mode), 'foo'; # this is better
$mode = 0644;   chmod $mode, 'foo';     # this is best
```
When above code is executed, it produces the following result:

### chomp

**Description**

This safer version of chop removes any trailing string that corresponds to the current value of $/ (also known as $INPUT_RECORD_SEPARATOR in the English module). It returns the total number of characters removed from all its arguments. By default $/ is set to new line character.

**Syntax**

Following is the simple syntax for this function:

```
chomp VARIABLE

chomp( LIST )

chomp
```

**Return Value**

This function returns Integer, number of bytes removed for all strings.

**Example**

Following is the example code showing its basic usage:

```
#!/usr/bin/perl

$string1 = "This is test";
$retval = chomp( $string1 );

print " Choped String is : $string1\n";
print " Number of characters removed : $retval\n";

$string1 = "This is test\n";
$retval = chomp( $string1 );
```
print " Choped String is : $string1
";
print " Number of characters removed : $retval
";

When above code is executed, it produces the following result:

Choped String is : This is test
Number of characters removed : 0
Choped String is : This is test
Number of characters removed : 1

chop

Description
This function removes the last character from EXPR, each element of LIST, or $._ if no value is specified.

Syntax
Following is the simple syntax for this function:

chop VARIABLE
chop( LIST )
chop

Return Value
This function returns the character removed from EXPR and in list context, the character is removed from the last element of LIST.

Example
Following is the example code showing its basic usage:

#!/usr/bin/perl

$string1 = "This is test";
$retval = chop( $string1 );
Perl

print " Choped String is : $string1\n";
print " Character removed : $retval\n";

When above code is executed, it produces the following result:

Choped String is : This is tes
Number of characters removed : t

chown

Description
This function changes the owner (and group) of a list of files. The first two elements of the list must be the numeric uid and gid, in that order. This function call works in similar way as unix command chown. Thus you should have sufficient privilege to change the permission of the file.

Syntax
Following is the simple syntax for this function:

```
chown USERID, GROUPID, LIST
```

Return Value
This function returns the number of files successfully changed.

Example
Following is the example code showing its basic usage:

```
#!/usr/bin/perl

$cnt = chown $uid, $gid, 'foo', 'bar';
chown $uid, $gid, @filenames;
```

When above code is executed, it produces the following result:
**chr**

**Description**
This function returns the character represented by the numeric value of EXPR, or \$_ if omitted, according to the current character set. Note that the character number will use the Unicode character numbers for numerical values above 127.

**Syntax**
Following is the simple syntax for this function:

```
chr EXPR
```

**Return Value**
This function returns the Character corresponding to the numeric value of EXPR.

**Example**
Following is the example code showing its basic usage:

```
#!/usr/bin/perl

print chr (35);
print "\n";
print chr (38);
print "\n";
print chr (40);
```

When above code is executed, it produces the following result:

```
#
&
(   
```

**NOTE:** You might be interested in ASCII Code Lookup Table.
ASCII Table Lookup

ASCII stands for American Standard Code for Information Interchange. There are 128 standard ASCII codes, each of which can be represented by a 7 digit binary number: 0000000 through 1111111.

Extended ASCII adds an additional 128 characters that vary between computers, programs and fonts.

7 Bit ASCII Codes

<table>
<thead>
<tr>
<th>DEC</th>
<th>OCT</th>
<th>HEX</th>
<th>BIN</th>
<th>Symbol</th>
<th>HTML Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>00000000</td>
<td>NUL</td>
<td>�</td>
<td>Null char</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>01</td>
<td>00000001</td>
<td>SOH</td>
<td>�</td>
<td>Start of Heading</td>
</tr>
<tr>
<td>2</td>
<td>002</td>
<td>02</td>
<td>00000010</td>
<td>STX</td>
<td>�</td>
<td>Start of Text</td>
</tr>
<tr>
<td>3</td>
<td>003</td>
<td>03</td>
<td>00000011</td>
<td>ETX</td>
<td>�</td>
<td>End of Text</td>
</tr>
<tr>
<td>4</td>
<td>004</td>
<td>04</td>
<td>00000100</td>
<td>EOT</td>
<td>�</td>
<td>End of Transmission</td>
</tr>
<tr>
<td>5</td>
<td>005</td>
<td>05</td>
<td>00000101</td>
<td>ENQ</td>
<td>�</td>
<td>Enquiry</td>
</tr>
<tr>
<td>6</td>
<td>006</td>
<td>06</td>
<td>00001110</td>
<td>ACK</td>
<td>�</td>
<td>Acknowledgment</td>
</tr>
<tr>
<td>7</td>
<td>007</td>
<td>07</td>
<td>00001111</td>
<td>BEL</td>
<td>�</td>
<td>Bell</td>
</tr>
<tr>
<td>8</td>
<td>010</td>
<td>08</td>
<td>00010000</td>
<td>BS</td>
<td>�</td>
<td>Back Space</td>
</tr>
<tr>
<td>9</td>
<td>011</td>
<td>09</td>
<td>00010001</td>
<td>HT</td>
<td>	</td>
<td>Horizontal Tab</td>
</tr>
<tr>
<td>10</td>
<td>012</td>
<td>0A</td>
<td>00010100</td>
<td>LF</td>
<td>
</td>
<td>Line Feed</td>
</tr>
<tr>
<td>11</td>
<td>013</td>
<td>0B</td>
<td>00010111</td>
<td>VT</td>
<td>�</td>
<td>Vertical Tab</td>
</tr>
<tr>
<td>12</td>
<td>014</td>
<td>0C</td>
<td>00011000</td>
<td>FF</td>
<td></td>
<td>Form Feed</td>
</tr>
<tr>
<td>13</td>
<td>015</td>
<td>0D</td>
<td>00011011</td>
<td>CR</td>
<td></td>
<td>Carriage Return</td>
</tr>
<tr>
<td>14</td>
<td>016</td>
<td>0E</td>
<td>00011110</td>
<td>SO</td>
<td>�</td>
<td>Shift Out / X-On</td>
</tr>
<tr>
<td>Number</td>
<td>Hexadecimal</td>
<td>ASCII</td>
<td>Description</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>-------</td>
<td>-------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>017 0F</td>
<td>00001111 SI</td>
<td>&amp;#015; Shift In / X-Off</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>020 10</td>
<td>00010000 DLE</td>
<td>&amp;#016; Data Line Escape</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>021 11</td>
<td>00010001 DC1</td>
<td>&amp;#017; Device Control 1 (oft. XON)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>022 12</td>
<td>00010010 DC2</td>
<td>&amp;#018; Device Control 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>023 13</td>
<td>00010011 DC3</td>
<td>&amp;#019; Device Control 3 (oft. XOFF)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>024 14</td>
<td>00010100 DC4</td>
<td>&amp;#020; Device Control 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>025 15</td>
<td>00010101 NAK</td>
<td>&amp;#021; Negative Acknowledgement</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>026 16</td>
<td>00010110 SYN</td>
<td>&amp;#022; Synchronous Idle</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>027 17</td>
<td>00010111 ETB</td>
<td>&amp;#023; End of Transmit Block</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>030 18</td>
<td>00011000 CAN</td>
<td>&amp;#024; Cancel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>031 19</td>
<td>00011001 EM</td>
<td>&amp;#025; End of Medium</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>032 1A</td>
<td>00011010 SUB</td>
<td>&amp;#026; Substitute</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>033 1B</td>
<td>00011011 ESC</td>
<td>&amp;#027; Escape</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>034 1C</td>
<td>00011100 FS</td>
<td>&amp;#028; File Separator</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>035 1D</td>
<td>00011101 GS</td>
<td>&amp;#029; Group Separator</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>036 1E</td>
<td>00011110 RS</td>
<td>&amp;#030; Record Separator</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>037 1F</td>
<td>00011111 US</td>
<td>&amp;#031; Unit Separator</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>040 20</td>
<td>00100000</td>
<td>&amp;#32; Space</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>041 21</td>
<td>00100001 !</td>
<td>&amp;#33; Exclamation mark</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>042 22</td>
<td>00100010 &quot;</td>
<td>&amp;#34; Double quotes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>043 23</td>
<td>00100011 #</td>
<td>&amp;#35; Number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hex</td>
<td>ASCII</td>
<td>Character</td>
<td>Hex</td>
<td>ASCII</td>
<td>Character</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-------</td>
<td>-------------</td>
<td>-----</td>
<td>-------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>044</td>
<td>24</td>
<td>00100100</td>
<td>$</td>
<td>$</td>
<td>Dollar</td>
</tr>
<tr>
<td>37</td>
<td>045</td>
<td>25</td>
<td>00100101</td>
<td>%</td>
<td>%</td>
<td>Procenttecken</td>
</tr>
<tr>
<td>38</td>
<td>046</td>
<td>26</td>
<td>00100110</td>
<td>&amp;</td>
<td>&amp;</td>
<td>Ampersand</td>
</tr>
<tr>
<td>39</td>
<td>047</td>
<td>27</td>
<td>00100111</td>
<td>'</td>
<td>'</td>
<td>Single quote</td>
</tr>
<tr>
<td>40</td>
<td>050</td>
<td>28</td>
<td>00101000</td>
<td>(</td>
<td>(</td>
<td>Open parenthesis</td>
</tr>
<tr>
<td>41</td>
<td>051</td>
<td>29</td>
<td>00101001</td>
<td>)</td>
<td>)</td>
<td>Close parenthesis</td>
</tr>
<tr>
<td>42</td>
<td>052</td>
<td>2A</td>
<td>00101010</td>
<td>*</td>
<td>*</td>
<td>Asterisk</td>
</tr>
<tr>
<td>43</td>
<td>053</td>
<td>2B</td>
<td>00101011</td>
<td>+</td>
<td>+</td>
<td>Plus</td>
</tr>
<tr>
<td>44</td>
<td>054</td>
<td>2C</td>
<td>00101100</td>
<td>,</td>
<td>,</td>
<td>Comma</td>
</tr>
<tr>
<td>45</td>
<td>055</td>
<td>2D</td>
<td>00101101</td>
<td>-</td>
<td>-</td>
<td>Hyphen</td>
</tr>
<tr>
<td>46</td>
<td>056</td>
<td>2E</td>
<td>00101110</td>
<td>.</td>
<td>.</td>
<td>Period, dot or full stop</td>
</tr>
<tr>
<td>47</td>
<td>057</td>
<td>2F</td>
<td>00101111</td>
<td>/</td>
<td>/</td>
<td>Slash or divide</td>
</tr>
<tr>
<td>48</td>
<td>060</td>
<td>30</td>
<td>00110000</td>
<td>0</td>
<td>0</td>
<td>Zero</td>
</tr>
<tr>
<td>49</td>
<td>061</td>
<td>31</td>
<td>00110001</td>
<td>1</td>
<td>1</td>
<td>One</td>
</tr>
<tr>
<td>50</td>
<td>062</td>
<td>32</td>
<td>00110010</td>
<td>2</td>
<td>2</td>
<td>Two</td>
</tr>
<tr>
<td>51</td>
<td>063</td>
<td>33</td>
<td>00110011</td>
<td>3</td>
<td>3</td>
<td>Three</td>
</tr>
<tr>
<td>52</td>
<td>064</td>
<td>34</td>
<td>00110100</td>
<td>4</td>
<td>4</td>
<td>Four</td>
</tr>
<tr>
<td>53</td>
<td>065</td>
<td>35</td>
<td>00110101</td>
<td>5</td>
<td>5</td>
<td>Five</td>
</tr>
<tr>
<td>54</td>
<td>066</td>
<td>36</td>
<td>00110110</td>
<td>6</td>
<td>6</td>
<td>Six</td>
</tr>
<tr>
<td>55</td>
<td>067</td>
<td>37</td>
<td>00110111</td>
<td>7</td>
<td>7</td>
<td>Seven</td>
</tr>
<tr>
<td>56</td>
<td>070</td>
<td>38</td>
<td>00111000</td>
<td>8</td>
<td>8</td>
<td>Eight</td>
</tr>
<tr>
<td>Decimal</td>
<td>Hex</td>
<td>ASCII</td>
<td>Character</td>
<td>Hex</td>
<td>ASCII</td>
<td>Character</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>--------</td>
<td>-----------</td>
<td>------</td>
<td>--------</td>
<td>-----------</td>
</tr>
<tr>
<td>57</td>
<td>071</td>
<td>39</td>
<td>00111001</td>
<td>9</td>
<td>9</td>
<td>Nine</td>
</tr>
<tr>
<td>58</td>
<td>072</td>
<td>3A</td>
<td>00111010</td>
<td>:</td>
<td>:</td>
<td>Colon</td>
</tr>
<tr>
<td>59</td>
<td>073</td>
<td>3B</td>
<td>00111011</td>
<td>;</td>
<td>;</td>
<td>Semicolon</td>
</tr>
<tr>
<td>60</td>
<td>074</td>
<td>3C</td>
<td>00111100</td>
<td>&lt;</td>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>61</td>
<td>075</td>
<td>3D</td>
<td>00111101</td>
<td>=</td>
<td>=</td>
<td>Equals</td>
</tr>
<tr>
<td>62</td>
<td>076</td>
<td>3E</td>
<td>00111110</td>
<td>&gt;</td>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>63</td>
<td>077</td>
<td>3F</td>
<td>00111111</td>
<td>?</td>
<td>?</td>
<td>Question mark</td>
</tr>
<tr>
<td>64</td>
<td>100</td>
<td>40</td>
<td>01000000</td>
<td>@</td>
<td>@</td>
<td>At symbol</td>
</tr>
<tr>
<td>65</td>
<td>101</td>
<td>41</td>
<td>01000001</td>
<td>A</td>
<td>A</td>
<td>Uppercase A</td>
</tr>
<tr>
<td>66</td>
<td>102</td>
<td>42</td>
<td>01000010</td>
<td>B</td>
<td>B</td>
<td>Uppercase B</td>
</tr>
<tr>
<td>67</td>
<td>103</td>
<td>43</td>
<td>01000011</td>
<td>C</td>
<td>C</td>
<td>Uppercase C</td>
</tr>
<tr>
<td>68</td>
<td>104</td>
<td>44</td>
<td>01000100</td>
<td>D</td>
<td>D</td>
<td>Uppercase D</td>
</tr>
<tr>
<td>69</td>
<td>105</td>
<td>45</td>
<td>01000101</td>
<td>E</td>
<td>E</td>
<td>Uppercase E</td>
</tr>
<tr>
<td>70</td>
<td>106</td>
<td>46</td>
<td>01000110</td>
<td>F</td>
<td>F</td>
<td>Uppercase F</td>
</tr>
<tr>
<td>71</td>
<td>107</td>
<td>47</td>
<td>01000111</td>
<td>G</td>
<td>G</td>
<td>Uppercase G</td>
</tr>
<tr>
<td>72</td>
<td>110</td>
<td>48</td>
<td>01001000</td>
<td>H</td>
<td>H</td>
<td>Uppercase H</td>
</tr>
<tr>
<td>73</td>
<td>111</td>
<td>49</td>
<td>01001001</td>
<td>I</td>
<td>I</td>
<td>Uppercase I</td>
</tr>
<tr>
<td>74</td>
<td>112</td>
<td>4A</td>
<td>01001010</td>
<td>J</td>
<td>J</td>
<td>Uppercase J</td>
</tr>
<tr>
<td>75</td>
<td>113</td>
<td>4B</td>
<td>01001011</td>
<td>K</td>
<td>K</td>
<td>Uppercase K</td>
</tr>
<tr>
<td>76</td>
<td>114</td>
<td>4C</td>
<td>01001100</td>
<td>L</td>
<td>L</td>
<td>Uppercase L</td>
</tr>
<tr>
<td>77</td>
<td>115</td>
<td>4D</td>
<td>01001101</td>
<td>M</td>
<td>M</td>
<td>Uppercase M</td>
</tr>
<tr>
<td>Dec</td>
<td>Hex</td>
<td>ASCII</td>
<td>Description</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-------</td>
<td>--------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>116</td>
<td>4E</td>
<td>01001110 N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>79</td>
<td>117</td>
<td>4F</td>
<td>01001111 O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>120</td>
<td>50</td>
<td>01010000 P</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>121</td>
<td>51</td>
<td>01010001 Q</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>82</td>
<td>122</td>
<td>52</td>
<td>01010010 R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>123</td>
<td>53</td>
<td>01010011 S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>124</td>
<td>54</td>
<td>01010100 T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>85</td>
<td>125</td>
<td>55</td>
<td>01010101 U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>86</td>
<td>126</td>
<td>56</td>
<td>01010110 V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>87</td>
<td>127</td>
<td>57</td>
<td>01010111 W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>130</td>
<td>58</td>
<td>01011000 X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>131</td>
<td>59</td>
<td>01011001 Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>132</td>
<td>5A</td>
<td>01011010 Z</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>91</td>
<td>133</td>
<td>5B</td>
<td>01011011 [</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>134</td>
<td>5C</td>
<td>01011100 \</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>93</td>
<td>135</td>
<td>5D</td>
<td>01011101 ]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>136</td>
<td>5E</td>
<td>01011110 ^</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>95</td>
<td>137</td>
<td>5F</td>
<td>01011111 _</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>140</td>
<td>60</td>
<td>01100000 `</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>97</td>
<td>141</td>
<td>61</td>
<td>01100001 a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>142</td>
<td>62</td>
<td>01100010 b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dec</td>
<td>Hex</td>
<td>ASCII</td>
<td>Binary</td>
<td>Hex Value</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-------</td>
<td>---------</td>
<td>-----------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>63</td>
<td>c</td>
<td>01100011</td>
<td>0x63</td>
<td>Lowercase c</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>64</td>
<td>d</td>
<td>01100100</td>
<td>0x64</td>
<td>Lowercase d</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>65</td>
<td>e</td>
<td>01100101</td>
<td>0x65</td>
<td>Lowercase e</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>66</td>
<td>f</td>
<td>01100110</td>
<td>0x66</td>
<td>Lowercase f</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>67</td>
<td>g</td>
<td>01100111</td>
<td>0x67</td>
<td>Lowercase g</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>68</td>
<td>h</td>
<td>01101000</td>
<td>0x68</td>
<td>Lowercase h</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>69</td>
<td>i</td>
<td>01101001</td>
<td>0x69</td>
<td>Lowercase i</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>6A</td>
<td>j</td>
<td>01101010</td>
<td>0x6A</td>
<td>Lowercase j</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>6B</td>
<td>k</td>
<td>01101011</td>
<td>0x6B</td>
<td>Lowercase k</td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>6C</td>
<td>l</td>
<td>01101100</td>
<td>0x6C</td>
<td>Lowercase l</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>6D</td>
<td>m</td>
<td>01101101</td>
<td>0x6D</td>
<td>Lowercase m</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>6E</td>
<td>n</td>
<td>01101110</td>
<td>0x6E</td>
<td>Lowercase n</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>6F</td>
<td>o</td>
<td>01101111</td>
<td>0x6F</td>
<td>Lowercase o</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>70</td>
<td>p</td>
<td>01110000</td>
<td>0x70</td>
<td>Lowercase p</td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>71</td>
<td>q</td>
<td>01110001</td>
<td>0x71</td>
<td>Lowercase q</td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>72</td>
<td>r</td>
<td>01110010</td>
<td>0x72</td>
<td>Lowercase r</td>
<td></td>
</tr>
<tr>
<td>115</td>
<td>73</td>
<td>s</td>
<td>01110011</td>
<td>0x73</td>
<td>Lowercase s</td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>74</td>
<td>t</td>
<td>01110100</td>
<td>0x74</td>
<td>Lowercase t</td>
<td></td>
</tr>
<tr>
<td>117</td>
<td>75</td>
<td>u</td>
<td>01110101</td>
<td>0x75</td>
<td>Lowercase u</td>
<td></td>
</tr>
<tr>
<td>118</td>
<td>76</td>
<td>v</td>
<td>01110110</td>
<td>0x76</td>
<td>Lowercase v</td>
<td></td>
</tr>
<tr>
<td>119</td>
<td>77</td>
<td>w</td>
<td>01110111</td>
<td>0x77</td>
<td>Lowercase w</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>OCT</td>
<td>HEX</td>
<td>BIN</td>
<td>Symbol</td>
<td>HTMLCode</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-------</td>
<td>------------</td>
<td>----------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>128</td>
<td>200</td>
<td>80</td>
<td>10000000</td>
<td>€</td>
<td>�</td>
<td>Euro sign</td>
</tr>
<tr>
<td>129</td>
<td>201</td>
<td>81</td>
<td>10000001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>202</td>
<td>82</td>
<td>10000010</td>
<td>,</td>
<td>�</td>
<td>Single low-9 quotation mark</td>
</tr>
<tr>
<td>131</td>
<td>203</td>
<td>83</td>
<td>10000011</td>
<td>f</td>
<td>�</td>
<td>Latin small letter f with hook</td>
</tr>
<tr>
<td>132</td>
<td>204</td>
<td>84</td>
<td>10000100</td>
<td>&quot;</td>
<td>�</td>
<td>Double low-9 quotation mark</td>
</tr>
<tr>
<td>133</td>
<td>205</td>
<td>85</td>
<td>10000101</td>
<td>…</td>
<td>�</td>
<td>Horizontal ellipsis</td>
</tr>
<tr>
<td>134</td>
<td>206</td>
<td>86</td>
<td>10000110</td>
<td>†</td>
<td>�</td>
<td>Dagger</td>
</tr>
<tr>
<td>135</td>
<td>207</td>
<td>87</td>
<td>10000111</td>
<td>‡</td>
<td>�</td>
<td>Double dagger</td>
</tr>
<tr>
<td>136</td>
<td>210</td>
<td>88</td>
<td>10001000</td>
<td>^</td>
<td>�</td>
<td>Modifier letter circumflex accent</td>
</tr>
</tbody>
</table>

Extended ASCII Codes

Below is set of additional 128 Extended ASCII Codes according to ISO 8859-1, also called ISO Latin-1.
<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Unicode</th>
<th>HTML Entity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>137</td>
<td>211</td>
<td>89</td>
<td>%0</td>
<td>Per mille sign</td>
</tr>
<tr>
<td>138</td>
<td>212</td>
<td>8A</td>
<td>Š</td>
<td>Latin capital letter S with caron</td>
</tr>
<tr>
<td>139</td>
<td>213</td>
<td>8B</td>
<td>&lt;</td>
<td>Single left-pointing angle quotation</td>
</tr>
<tr>
<td>140</td>
<td>214</td>
<td>8C</td>
<td>Ø</td>
<td>Latin capital ligature OE</td>
</tr>
<tr>
<td>141</td>
<td>215</td>
<td>8D</td>
<td></td>
<td>10001101</td>
</tr>
<tr>
<td>142</td>
<td>216</td>
<td>8E</td>
<td>Ž</td>
<td>Latin capital letter Z with caron</td>
</tr>
<tr>
<td>143</td>
<td>217</td>
<td>8F</td>
<td></td>
<td>10001111</td>
</tr>
<tr>
<td>144</td>
<td>220</td>
<td>90</td>
<td></td>
<td>10010000</td>
</tr>
<tr>
<td>145</td>
<td>221</td>
<td>91</td>
<td>‘</td>
<td>Left single quotation mark</td>
</tr>
<tr>
<td>146</td>
<td>222</td>
<td>92</td>
<td>’</td>
<td>Right single quotation mark</td>
</tr>
<tr>
<td>147</td>
<td>223</td>
<td>93</td>
<td>“</td>
<td>Left double quotation mark</td>
</tr>
<tr>
<td>148</td>
<td>224</td>
<td>94</td>
<td>”</td>
<td>Right double quotation mark</td>
</tr>
<tr>
<td>149</td>
<td>225</td>
<td>95</td>
<td>•</td>
<td>Bullet</td>
</tr>
<tr>
<td>150</td>
<td>226</td>
<td>96</td>
<td>–</td>
<td>En dash</td>
</tr>
<tr>
<td>151</td>
<td>227</td>
<td>97</td>
<td>—</td>
<td>Em dash</td>
</tr>
<tr>
<td>152</td>
<td>230</td>
<td>98</td>
<td>~</td>
<td>Small tilde</td>
</tr>
<tr>
<td>153</td>
<td>231</td>
<td>99</td>
<td>™</td>
<td>Trade mark sign</td>
</tr>
<tr>
<td>154</td>
<td>232</td>
<td>9A</td>
<td>š</td>
<td>Latin small letter S with caron</td>
</tr>
<tr>
<td>155</td>
<td>233</td>
<td>9B</td>
<td>&gt;</td>
<td>Single right-pointing angle quotation mark</td>
</tr>
<tr>
<td>Unicode</td>
<td>Character</td>
<td>Hex</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>----------------</td>
<td>--------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>156</td>
<td>234 9C</td>
<td>10011100 œ</td>
<td>Latin small ligature oe</td>
<td></td>
</tr>
<tr>
<td>157</td>
<td>235 9D</td>
<td>10011101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>158</td>
<td>236 9E</td>
<td>10011110 ź</td>
<td>Latin small letter z with caron</td>
<td></td>
</tr>
<tr>
<td>159</td>
<td>237 9F</td>
<td>10011111 Ŷ</td>
<td>Latin capital letter Y with diaeresis</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>240 A0</td>
<td>10100000 &amp;</td>
<td>Non-breaking space</td>
<td></td>
</tr>
<tr>
<td>161</td>
<td>241 A1</td>
<td>10100001 i</td>
<td>Inverted exclamation mark</td>
<td></td>
</tr>
<tr>
<td>162</td>
<td>242 A2</td>
<td>10100010 ₤</td>
<td>Cent sign</td>
<td></td>
</tr>
<tr>
<td>163</td>
<td>243 A3</td>
<td>10100011 £</td>
<td>Pound sign</td>
<td></td>
</tr>
<tr>
<td>164</td>
<td>244 A4</td>
<td>10100100 ¦</td>
<td>Currency sign</td>
<td></td>
</tr>
<tr>
<td>165</td>
<td>245 A5</td>
<td>10100101 ¥</td>
<td>Yen sign</td>
<td></td>
</tr>
<tr>
<td>166</td>
<td>246 A6</td>
<td>10100110 ‖</td>
<td>Pipe, Broken vertical bar</td>
<td></td>
</tr>
<tr>
<td>167</td>
<td>247 A7</td>
<td>10100111 §</td>
<td>Section sign</td>
<td></td>
</tr>
<tr>
<td>168</td>
<td>250 A8</td>
<td>10101000 &quot;</td>
<td>Spacing diaeresis - umlaut</td>
<td></td>
</tr>
<tr>
<td>169</td>
<td>251 A9</td>
<td>10101001 ©</td>
<td>Copyright sign</td>
<td></td>
</tr>
<tr>
<td>170</td>
<td>252 AA</td>
<td>10101010 a</td>
<td>Feminine ordinal indicator</td>
<td></td>
</tr>
<tr>
<td>171</td>
<td>253 AB</td>
<td>10101011 «</td>
<td>Left double angle quotes</td>
<td></td>
</tr>
<tr>
<td>172</td>
<td>254 AC</td>
<td>10101100 ¬</td>
<td>Not sign</td>
<td></td>
</tr>
<tr>
<td>173</td>
<td>255 AD</td>
<td>10101101</td>
<td>Soft hyphen</td>
<td></td>
</tr>
<tr>
<td>174</td>
<td>256 AE</td>
<td>10101110 ®</td>
<td>Registered trade mark sign</td>
<td></td>
</tr>
<tr>
<td>175</td>
<td>257 AF</td>
<td>10101111 ‒</td>
<td>Spacing macron - overline</td>
<td></td>
</tr>
<tr>
<td>Unicode</td>
<td>Decimal</td>
<td>Character</td>
<td>HTML Entity</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>-------------</td>
<td>-------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>176</td>
<td>260 B0</td>
<td>10110000</td>
<td>°</td>
<td>Degree sign</td>
</tr>
<tr>
<td>177</td>
<td>261 B1</td>
<td>10110001</td>
<td>±</td>
<td>Plus-or-minus sign</td>
</tr>
<tr>
<td>178</td>
<td>262 B2</td>
<td>10110010</td>
<td>²</td>
<td>Superscript two - squared</td>
</tr>
<tr>
<td>179</td>
<td>263 B3</td>
<td>10110011</td>
<td>³</td>
<td>Superscript three - cubed</td>
</tr>
<tr>
<td>180</td>
<td>264 B4</td>
<td>10110100</td>
<td>′</td>
<td>Acute accent - spacing acute</td>
</tr>
<tr>
<td>181</td>
<td>265 B5</td>
<td>10110101</td>
<td>µ</td>
<td>Micro sign</td>
</tr>
<tr>
<td>182</td>
<td>266 B6</td>
<td>10110110</td>
<td>¶</td>
<td>Pilcrow sign - paragraph sign</td>
</tr>
<tr>
<td>183</td>
<td>267 B7</td>
<td>10110111</td>
<td>·</td>
<td>Middle dot - Georgian comma</td>
</tr>
<tr>
<td>184</td>
<td>270 B8</td>
<td>10111000</td>
<td>`,</td>
<td>Spacing cedilla</td>
</tr>
<tr>
<td>185</td>
<td>271 B9</td>
<td>10111001</td>
<td>¹</td>
<td>Superscript one</td>
</tr>
<tr>
<td>186</td>
<td>272 BA</td>
<td>10111010</td>
<td>₀</td>
<td>Masculine ordinal indicator</td>
</tr>
<tr>
<td>187</td>
<td>273 BB</td>
<td>10111011</td>
<td>»</td>
<td>Right double angle quotes</td>
</tr>
<tr>
<td>188</td>
<td>274 BC</td>
<td>10111100</td>
<td>¼</td>
<td>Fraction one quarter</td>
</tr>
<tr>
<td>189</td>
<td>275 BD</td>
<td>10111101</td>
<td>½</td>
<td>Fraction one half</td>
</tr>
<tr>
<td>190</td>
<td>276 BE</td>
<td>10111110</td>
<td>¾</td>
<td>Fraction three quarters</td>
</tr>
<tr>
<td>191</td>
<td>277 BF</td>
<td>10111111</td>
<td>¿</td>
<td>Inverted question mark</td>
</tr>
<tr>
<td>192</td>
<td>300 C0</td>
<td>11000000</td>
<td>À</td>
<td>Latin capital letter A with grave</td>
</tr>
<tr>
<td>193</td>
<td>301 C1</td>
<td>11000001</td>
<td>Á</td>
<td>Latin capital letter A with acute</td>
</tr>
<tr>
<td>194</td>
<td>302 C2</td>
<td>11000010</td>
<td>Â</td>
<td>Latin capital letter A with circumflex</td>
</tr>
<tr>
<td>Dec</td>
<td>Hex</td>
<td>Char</td>
<td>ASCII Code</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-------</td>
<td>------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>195</td>
<td>303</td>
<td>C3</td>
<td>11000011</td>
<td>Å</td>
</tr>
<tr>
<td>196</td>
<td>304</td>
<td>C4</td>
<td>11001000</td>
<td>À</td>
</tr>
<tr>
<td>197</td>
<td>305</td>
<td>C5</td>
<td>11001011</td>
<td>Å</td>
</tr>
<tr>
<td>198</td>
<td>306</td>
<td>C6</td>
<td>11001100</td>
<td>Æ</td>
</tr>
<tr>
<td>199</td>
<td>307</td>
<td>C7</td>
<td>11001111</td>
<td>Ç</td>
</tr>
<tr>
<td>200</td>
<td>310</td>
<td>C8</td>
<td>11010000</td>
<td>È</td>
</tr>
<tr>
<td>201</td>
<td>311</td>
<td>C9</td>
<td>11010001</td>
<td>Ë</td>
</tr>
<tr>
<td>202</td>
<td>312</td>
<td>CA</td>
<td>11010100</td>
<td>È</td>
</tr>
<tr>
<td>203</td>
<td>313</td>
<td>CB</td>
<td>11010101</td>
<td>Ë</td>
</tr>
<tr>
<td>204</td>
<td>314</td>
<td>CC</td>
<td>11011000</td>
<td>Ì</td>
</tr>
<tr>
<td>205</td>
<td>315</td>
<td>CD</td>
<td>11011011</td>
<td>Í</td>
</tr>
<tr>
<td>206</td>
<td>316</td>
<td>CE</td>
<td>11011110</td>
<td>Ì</td>
</tr>
<tr>
<td>207</td>
<td>317</td>
<td>CF</td>
<td>11011111</td>
<td>Ì</td>
</tr>
<tr>
<td>208</td>
<td>320</td>
<td>D0</td>
<td>11010000</td>
<td>Ð</td>
</tr>
<tr>
<td>209</td>
<td>321</td>
<td>D1</td>
<td>11010001</td>
<td>Ñ</td>
</tr>
<tr>
<td>Code</td>
<td>Value</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>210</td>
<td>322</td>
<td>D2 11010010 Ö &amp; #210; Latin capital letter O with grave</td>
<td></td>
<td></td>
</tr>
<tr>
<td>211</td>
<td>323</td>
<td>D3 11010011 Ö &amp; #211; Latin capital letter O with acute</td>
<td></td>
<td></td>
</tr>
<tr>
<td>212</td>
<td>324</td>
<td>D4 11010100 Ö &amp; #212; Latin capital letter O with circumflex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>213</td>
<td>325</td>
<td>D5 11010101 Ö &amp; #213; Latin capital letter O with tilde</td>
<td></td>
<td></td>
</tr>
<tr>
<td>214</td>
<td>326</td>
<td>D6 11010110 Ö &amp; #214; Latin capital letter O with diaeresis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>215</td>
<td>327</td>
<td>D7 11010111 × &amp; #215; Multiplication sign</td>
<td></td>
<td></td>
</tr>
<tr>
<td>216</td>
<td>330</td>
<td>D8 11011000 Ø &amp; #216; Latin capital letter O with slash</td>
<td></td>
<td></td>
</tr>
<tr>
<td>217</td>
<td>331</td>
<td>D9 11011001 Ú &amp; #217; Latin capital letter U with grave</td>
<td></td>
<td></td>
</tr>
<tr>
<td>218</td>
<td>332</td>
<td>DA 11011010 Ú &amp; #218; Latin capital letter U with acute</td>
<td></td>
<td></td>
</tr>
<tr>
<td>219</td>
<td>333</td>
<td>DB 11011011 Û &amp; #219; Latin capital letter U with circumflex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>220</td>
<td>334</td>
<td>DC 11011100 Ù &amp; #220; Latin capital letter U with diaeresis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>221</td>
<td>335</td>
<td>DD 11011101 Ý &amp; #221; Latin capital letter Y with acute</td>
<td></td>
<td></td>
</tr>
<tr>
<td>222</td>
<td>336</td>
<td>DE 11011110 Þ &amp; #222; Latin capital letter THORN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>223</td>
<td>337</td>
<td>DF 11011111 ß &amp; #223; Latin small letter sharp s - ess-zed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>224</td>
<td>340</td>
<td>E0 11100000 à &amp; #224; Latin small letter a with grave</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Dec</td>
<td>Hex</td>
<td>Binary</td>
<td>Unicode Char</td>
</tr>
<tr>
<td>------</td>
<td>-----</td>
<td>-----</td>
<td>--------</td>
<td>--------------</td>
</tr>
<tr>
<td>225</td>
<td>341</td>
<td>E1</td>
<td>11100001</td>
<td>á</td>
</tr>
<tr>
<td>226</td>
<td>342</td>
<td>E2</td>
<td>11100010</td>
<td>à</td>
</tr>
<tr>
<td>227</td>
<td>343</td>
<td>E3</td>
<td>11100011</td>
<td>ā</td>
</tr>
<tr>
<td>228</td>
<td>344</td>
<td>E4</td>
<td>11100100</td>
<td>ā</td>
</tr>
<tr>
<td>229</td>
<td>345</td>
<td>E5</td>
<td>11100101</td>
<td>å</td>
</tr>
<tr>
<td>230</td>
<td>346</td>
<td>E6</td>
<td>11100110</td>
<td>æ</td>
</tr>
<tr>
<td>231</td>
<td>347</td>
<td>E7</td>
<td>11100111</td>
<td>ç</td>
</tr>
<tr>
<td>232</td>
<td>350</td>
<td>E8</td>
<td>11101000</td>
<td>è</td>
</tr>
<tr>
<td>233</td>
<td>351</td>
<td>E9</td>
<td>11101001</td>
<td>é</td>
</tr>
<tr>
<td>234</td>
<td>352</td>
<td>EA</td>
<td>11101010</td>
<td>è</td>
</tr>
<tr>
<td>235</td>
<td>353</td>
<td>EB</td>
<td>11101011</td>
<td>ë</td>
</tr>
<tr>
<td>236</td>
<td>354</td>
<td>EC</td>
<td>11101100</td>
<td>ï</td>
</tr>
<tr>
<td>237</td>
<td>355</td>
<td>ED</td>
<td>11101101</td>
<td>ï</td>
</tr>
<tr>
<td>238</td>
<td>356</td>
<td>EE</td>
<td>11101110</td>
<td>î</td>
</tr>
<tr>
<td>239</td>
<td>357</td>
<td>EF</td>
<td>11101111</td>
<td>ï</td>
</tr>
<tr>
<td>Unicode</td>
<td>Dec</td>
<td>Hex</td>
<td>Character</td>
<td>HTML Entity</td>
</tr>
<tr>
<td>---------</td>
<td>-----</td>
<td>-----</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>240</td>
<td>360</td>
<td>F0</td>
<td>ò</td>
<td>ð</td>
</tr>
<tr>
<td>241</td>
<td>361</td>
<td>F1</td>
<td>ñ</td>
<td>ñ</td>
</tr>
<tr>
<td>242</td>
<td>362</td>
<td>F2</td>
<td>õ</td>
<td>ò</td>
</tr>
<tr>
<td>243</td>
<td>363</td>
<td>F3</td>
<td>ó</td>
<td>ó</td>
</tr>
<tr>
<td>244</td>
<td>364</td>
<td>F4</td>
<td>ô</td>
<td>ô</td>
</tr>
<tr>
<td>245</td>
<td>365</td>
<td>F5</td>
<td>ö</td>
<td>õ</td>
</tr>
<tr>
<td>246</td>
<td>366</td>
<td>F6</td>
<td>ò</td>
<td>ö</td>
</tr>
<tr>
<td>247</td>
<td>367</td>
<td>F7</td>
<td>÷</td>
<td>÷</td>
</tr>
<tr>
<td>248</td>
<td>368</td>
<td>F8</td>
<td>ø</td>
<td>ø</td>
</tr>
<tr>
<td>249</td>
<td>369</td>
<td>F9</td>
<td>ù</td>
<td>ù</td>
</tr>
<tr>
<td>250</td>
<td>370</td>
<td>FA</td>
<td>ū</td>
<td>ú</td>
</tr>
<tr>
<td>251</td>
<td>371</td>
<td>FB</td>
<td>ų</td>
<td>û</td>
</tr>
<tr>
<td>252</td>
<td>372</td>
<td>FC</td>
<td>ü</td>
<td>ü</td>
</tr>
<tr>
<td>253</td>
<td>373</td>
<td>FD</td>
<td>ý</td>
<td>ý</td>
</tr>
<tr>
<td>254</td>
<td>374</td>
<td>FE</td>
<td>þ</td>
<td>þ</td>
</tr>
<tr>
<td>255</td>
<td>375</td>
<td>FF</td>
<td>ý</td>
<td>ÿ</td>
</tr>
</tbody>
</table>
chroot

Description
This function works like the system call by the same name: it makes the named directory the new root directory for all further pathnames that begin with a / by your process and all its children. For security reasons, this function, which is identical to the system chroot() function, is restricted to the superuser and cannot be undone.

If FILENAME is omitted, then it does a chroot to $.

Syntax
Following is the simple syntax for this function:

```
chroot EXPR
```

Return Value
This function returns 0 on failure and 1 on success.

Example
Following is the example code showing its basic usage:

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

When above code is executed, it produces the following result:

close

Description
This function closes FILEHANDLE, flushing the buffers, if appropriate, and disassociating the FILEHANDLE with the original file, pipe, or socket. Closes the currently selected filehandle if none is specified.
**Syntax**

Following is the simple syntax for this function:

```
close FILEHANDLE
```

**Return Value**

This function returns 0 on failure and 1 if buffers were flushed and the file was successfully closed.

**Example**

Following is the example code showing its basic usage:

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

When above code is executed, it produces the following result:

```
closedir
```

**closedir**

**Description**

This function closes the directory handle DIRHANDLE.

**Syntax**

Following is the simple syntax for this function:

```
closedir DIRHANDLE
```

**Return Value**

This function returns 0 on failure and 1 on success.

**Example**

Following is the example code showing its basic usage:

```
#!/usr/bin/perl -w
```
$dirname = "/tmp";
open( DIR, $dirname ) || die "Error in opening dir $dirname\n";
while( ($filename = readdir(DIR)) ){
    print("$filename\n");
}
closedir(DIR);

When above code is executed, it produces the following result:

```
.
..  
testdir
```

### connect

**Description**

This function connects to the remote socket using the filehandle SOCKET and the address specified by EXPR. The EXPR should be a packed address of the appropriate type for the socket.

**Syntax**

Following is the simple syntax for this function:

```
connect SOCKET, EXPR
```

**Return Value**

This function returns 0 on failure and 1 on success.

**Example**

Following is the example code showing its basic usage:

When above code is executed, it produces the following result:
**continue**

**Description**

This function is a flow control statement rather than a function. If there is a `continue BLOCK` attached to a `BLOCK` (typically in a `while` or `foreach`), it is always executed just before the conditional is about to be evaluated again, just like the third part of a `for` loop in C.

Thus it can be used to increment a loop variable, even when the loop has been continued via the `next` statement. `last, next, or redo` may appear within a `continue block`.

**Syntax**

Following is the simple syntax for this function:

```
continue BLOCK
```

**Return Value**

This function does not return anything.

**Example**

Following is the example code showing its basic usage:

```
while (EXPR) {
    ### redo always comes here
    do_something;
} continue {
    ### next always comes here
    do_something_else;
    # then back the top to re-check EXPR
}  
### last always comes here
```

When above code is executed, it produces the following result:
**cos**

**Description**
This function returns the cosine of EXPR, or \$_ if EXPR is omitted. The value should be expressed in radians.

**Syntax**
Following is the simple syntax for this function:

```
  cos EXPR

  cos
```

**Return Value**
This function returns floating point number.

**Example**
Following is the example code showing its basic usage:

```

```
When above code is executed, it produces the following result:

```

```

**crypt**

**Description**
This function encrypts the string EXPR using the system crypt( ) function. The value of SALT is used to select an encrypted version from one of a number of variations.

You cannot decrypt a string that has been encrypted in this way. It's normally used one way, first to encrypt a string, and then to encrypt a password to compare against the encrypted string. If you're using it in this form, then consider supplying the encrypted password as the SALT.

**Syntax**
Following is the simple syntax for this function:
crypt EXPR, SALT

**Return Value**
This function returns the encrypted string.

**Example**
Following is the example code showing its basic usage, it makes sure that whoever runs this program knows their password:

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

$pwd = (getpwuid($<))[1];

system "stty -echo";
print "Password: ";
chomp($word = <STDIN>);
print "\n";
system "stty echo";

if (crypt($word, $pwd) ne $pwd) {
    die "Sorry wrong password\n";
} else {
    print "ok, correct password\n";
}
```

When above code is executed, it produces the following result:

**dbmclose**

**Description**
This function closes the binding between a hash and a DBM file. Use the tie function with a suitable module.
Perl

Syntax
Following is the simple syntax for this function:

```
dbmclose HASH
```

Return Value
This function returns 0 on failure and 1 on success.

Note that functions such as keys and values may return huge lists when used on large DBM files. You may prefer to use the each function to iterate over large DBM files.

Example
Following is the example code showing its basic usage:

```
#!/usr/bin/perl

# print out history file offsets
dbmopen(%HIST,'/usr/lib/news/history',0666);
while ((($key,$val) = each %HIST) {
  print $key, ' = ', unpack('L',$val), "\n";
}
dbmclose(%HIST);
```

When above code is executed, it produces the following result:

```
```

dbmopen

Description
This function Binds the database file specified by EXPR to the hash HASH. If the database does not exist, then it is created using the mode specified by MODE. The file EXPR should be specified without the .dir and .pag extensions. Use is now deprecated in favor of tie and one of the tied DBM hash modules, such as SDBM_File.

Syntax
Following is the simple syntax for this function:
dbmopen HASH, EXPR, MODE

**Return Value**

This function returns 0 on failure and 1 on success.

Note that functions such as *keys* and *values* may return huge lists when used on large DBM files. You may prefer to use the each function to iterate over large DBM files.

**Example**

Following is the example code showing its basic usage:

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

# print out history file offsets
$dbmopen(%HIST,'/usr/lib/news/history',0666);
while (($key,$val) = each %HIST) {
    print $key, ' = ', unpack('L',$val), "\n";
}
dbmclose(%HIST);
```

When above code is executed, it produces the following result:

```
defined
```

**defined**

**Description**

This function returns true if *EXPR* has a value other than the undef value, or checks the value of `_` if *EXPR* is not specified. This can be used with many functions to detect a failure in operation, since they return undef if there was a problem. A simple Boolean test does not differentiate between false, zero, an empty string, or the string `.0.,` which are all equally false.

If *EXPR* is a function or function reference, then it returns true if the function has been defined. When used with entire arrays and hashes, it will not always produce intuitive results. If a hash element is specified, it returns true if the corresponding value has been defined, but does not determine whether the specified key exists in the hash.
**Syntax**

Following is the simple syntax for this function:

```
defined EXPR

defined
```

**Return Value**

This function returns 0 if `EXPR` contains `undef` and 1 if `EXPR` contains a valid value or reference.

**Example**

Following is the example code showing its basic usage:

```
#!/usr/bin/perl

$var1 = "This is defined";

if( defined($var1) ){
    print "$var1\n";
} else{
    print "var2 is not defined\n";
}
```

When above code is executed, it produces the following result:

```
This is defined
var2 is not defined
```
delete

Description
This function deletes the specified keys and associated values from a hash, or the specified elements from an array. The operation works on individual elements or slices.

Syntax
Following is the simple syntax for this function:

```perl
delete LIST
```

Return Value
This function returns undef if the key does not exist and value associated with the deleted hash key or array index.

Example
Following is the example code showing its basic usage, it deletes all the values of %HASH and @ARRAY:

```perl
When above code is executed, it produces the following result:
```

die

Description
This function prints the value of LIST to STDERR and calls exit with the error value contained in $!.

Syntax
Following is the simple syntax for this function:

```perl
die LIST
```
**Return Value**
This function does not return any value.

**Example**
Following is the example code showing its basic usage:

```perl
do BLOCK

do EXPR

do SUB(LIST)
```

When above code is executed, it produces the following result:

**do**

**Description**
This function when supplied a block, do executes as if BLOCK were a function, returning the value of the last statement evaluated in the block.

When supplied with EXPR, do executes the file specified by EXPR as if it were another Perl script.

If supplied a subroutine, SUB, do executes the subroutine using LIST as the arguments, raising an exception if SUB hasn't been defined.

**Syntax**
Following is the simple syntax for this function:

**Return Value**
This function returns the value of the last statement evaluated in the block.

**Example**
Following is the example code showing its basic usage:
dump

Description
This function Dumps the currently executing Perl interpreter and script into a core dump. Using the undump program, you can then reconstitute the dumped core into an executable program.

When the new binary is executed it will begin by executing a goto LABEL. If LABEL is omitted, restarts the program from the top.

If you're looking to use dump to speed up your program, consider generating bytecode or native C code as described in perlcc. The perlcc generates executables from Perl programs and this compiler comes alongwith PERL installation.

Syntax
Following is the simple syntax for this function:

```
dump LABEL

dump
```

Return Value
This function does not return any value.

Example
Following is the example code showing its basic usage:

```

```

When above code is executed, it produces the following result:
**each**

**Description**
This function when called in list context, returns a 2-element list consisting of the key and value for the next element of a hash, so that you can iterate over it. When called in scalar context, returns only the key for the next element in the hash.

**Syntax**
Following is the simple syntax for this function:

```
each HASH
```

**Return Value**
This function when called in list context, returns a 2-element list consisting of the key and value for the next element of a hash, so that you can iterate over it. It returns only the key for the next element in the hash when called in scalar context.

**Example**
Following is the example code showing its basic usage, this will print out all environment variables.: 

When above code is executed, it produces the following result:

**endgrent**

**Description**
This function tells the system you no longer expect to read entries from the groups file using getgrent.

**Syntax**
Following is the simple syntax for this function:

```
endgrent
```
**Return Value**

This function does not return any value.

**Example**

Following is the example code showing its basic usage:

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

while( ($name, $aliases, $addrtype, $length, @addrs) = gethostent() ){
    print "Name  = $name\n";
    print "Aliases  = $aliases\n";
    print "Addr Type  = $addrtype\n";
    print "Length  = $length\n";
    print "Addrs  = @addrs\n";
}
```

When above code is executed, it produces the following result:

**endhostent**

**Description**

This function tells the system you no longer expect to read entries from the hosts file using gethostent.

**Syntax**

Following is the simple syntax for this function:

```
endhostent
```

**Return Value**

This function does not return any value.

**Example**

Following is the example code showing its basic usage:
sethostent(1);

while( ($name, $aliases, $addrtype, $length, @addrs) = gethostent() ){
    print "Name  = $name\n";
    print "Aliases = $aliases\n";
    print "Addr Type  = $addrtype\n";
    print "Length  = $length\n";
    print "Addrs  = @addrs\n";
}

endhostent();  # Closes the database;

When above code is executed, it produces the following result:

Name  = ip-50-62-147-141.ip.secureserver.net
Aliases = ip-50-62-147-141 localhost.secureserver.net
localhost.localdomain localhost
Addr Type  = 2
Length  = 4
Addrs  =

Name  = ip-50-62-147-141.ip.secureserver.net
Aliases = ip-50-62-147-141 localhost.secureserver.net
localhost.localdomain localhost
Addr Type  = 2
Length  = 4
Addrs  =

endnetent

Description

This function tells the system you no longer expect to read entries from the networks list using getnetent.
Syntax
Following is the simple syntax for this function:

```
endnetent
```

Return Value
This function does not return any value.

Example
Following is the example code showing its basic usage:

```
#!/usr/bin/perl

use Socket;

while ( ($name, $aliases, $addrtype, $net) = getnetent() ){
    print "Name = $name\n";
    print "Aliases = $aliases\n";
    print "Addrtype = $addrtype\n";
    print "Net = $net\n";
}

setnetent(1); # Rewind the database;

while ( ($name, $aliases, $addrtype, $net) = getnetent() ){
    print "Name = $name\n";
    print "Aliases = $aliases\n";
    print "Addrtype = $addrtype\n";
    print "Net = $net\n";
}

endnetent(); # Closes the database;
```

When above code is executed, it produces the following result:
endprotoent

Description
This function tells the system you no longer expect to read entries from the
protocols list using getprotoent.

Syntax
Following is the simple syntax for this function:
endprotoent

**Return Value**
This function does not return any value.

**Example**
Following is the example code showing its basic usage:

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

while((@$name, @$aliases, @$protocol_number) = getprotoent()){  
    print "Name = $name\n";
    print "Aliases = $aliases\n";
    print "Protocol Number = $protocol_number\n";
}

setprotoent(1); # Rewind the database.

while((@$name, @$aliases, @$protocol_number) = getprotoent()){  
    print "Name = $name\n";
    print "Aliases = $aliases\n";
    print "Protocol Number = $protocol_number\n";
}
endprotoent(); # Closes the database
```

When above code is executed, it produces the following result:

```
Name = ip
Aliases = IP
Protocol Number = 0
Name = hopopt
Aliases = HOPOPT
Protocol Number = 0
Name = icmp
Aliases = ICMP
Protocol Number = 1
```
Name = igmp  
Aliases = IGMP  
Protocol Number = 2

Name = ggp  
Aliases = GGP  
Protocol Number = 3

Name = ipencap  
Aliases = IP-ENCAP  
Protocol Number = 4

Name = st  
Aliases = ST  
Protocol Number = 5

Name = tcp  
Aliases = TCP  
Protocol Number = 6

Name = manet  
Aliases = manet  
Protocol Number = 138

Name = hip  
Aliases = HIP  
Protocol Number = 139

Name = shim6  
Aliases = Shim6  
Protocol Number = 140

---

**endpwent**

**Description**

This function tells the system you no longer expect to read entries from the password file using getpwent. Under Windows, use the Win32API::Net function to get the information from a domain server.
Syntax
Following is the simple syntax for this function:

```
endpwent
```

Return Value
This function does not return any value.

Example
Following is the example code showing its basic usage:

```
#!/usr/bin/perl

while((
    $name,
    $passwd,
    $uid,
    $gid,
    $quota,
    $comment,
    $gcos,
    $dir,
    $shell) = getpwent()){
    print "Name = $name\n";
    print "Password = $passwd\n";
    print "UID = $uid\n";
    print "GID = $gid\n";
    print "Quota = $quota\n";
    print "Comment = $comment\n";
    print "Gcos = $gcos\n";
    print "HOME DIR = $dir\n";
    print "Shell = $shell\n";
}

setpwent(); # Rewind the database /etc/passwd

while((
    $name,
    $passwd,
    $uid,
    $gid,
    $quota,
    $comment,
    $gcos,
    $dir,
    $shell) = getpwent()){
    print "Name = $name\n";
    print "Password = $passwd\n";
    print "UID = $uid\n";
    print "GID = $gid\n";
    print "Quota = $quota\n";
```
perl

```perl
print "Comment = $comment\n";
print "Gcos = $gcos\n";
print "HOME DIR = $dir\n";
print "Shell = $shell\n";
}

derpwent(); # Closes the database;
```

When above code is executed, it produces the following result:

Name = root
Password = x
UID = 0
GID = 0
Quota =
Comment =
Gcos = root
HOME DIR = /root
Shell = /bin/bash

Name = bin
Password = x
UID = 1
GID = 1
Quota =
Comment =
Gcos = bin
HOME DIR = /bin
Shell = /sbin/nologin

Name = daemon
Password = x
UID = 2
GID = 2
Quota =
Comment =
Gcos = daemon
HOME DIR = /sbin
Shell = /sbin/nologin
Name = adm
Password = x
UID = 3
GID = 4
Quota =
Comment =
Gcos = adm
HOME DIR = /var/adm
Shell = /sbin/nologin
Name = lp
Password = x
UID = 4
GID = 7
Quota =
Comment =
Gcos = lp
HOME DIR = /var/spool/lpd
Shell = /sbin/nologin
Name = sync
Password = x
UID = 5
GID = 0
Quota =
Comment =
Gcos = sync
HOME DIR = /sbin
Shell = /bin/sync
Name = shutdown
Password = x
UID = 6
GID = 0
Quota =
Comment =
Gcos = shutdown
HOME DIR = /sbin
Shell = /sbin/shutdown
Name = halt
Password = x
UID = 7
GID = 0
Quota =
Comment =
Gcos = halt
HOME DIR = /sbin
Shell = /sbin/halt
Name = mail
Password = x
UID = 8
GID = 12
Quota =
Comment =
Gcos = mail
HOME DIR = /var/spool/mail
Shell = /sbin/nologin
Name = uucp
Password = x
UID = 10
GID = 14
Quota =
Comment =
Gcos = uucp
HOME DIR = /var/spool/uucp
Shell = /sbin/nologin
Name = operator
Password = x
UID = 11
GID = 0
Quota =
Comment =
Gcos = operator
HOME DIR = /root
Shell = /sbin/nologin
Name = games
Password = x
UID = 12
GID = 100
Quota =
Comment =
Gcos = games
HOME DIR = /usr/games
Shell = /sbin/nologin
Name = gopher
Password = x
UID = 13
GID = 30
Quota =
Comment =
Gcos = gopher
HOME DIR = /var/gopher
Shell = /sbin/nologin
Name = ftp
Password = x
UID = 14
GID = 50
Quota =
Comment =
Gcos = FTP User
HOME DIR = /var/ftp
Shell = /sbin/nologin
Name = nobody
Password = x
UID = 99
GID = 99
Quota =
Comment =
Gcos = Nobody
HOME DIR = /
Shell = /sbin/nologin
Name = dbus
Password = x
UID = 81
GID = 81
Quota =
Comment =
Gcos = System message bus
HOME DIR = /
Shell = /sbin/nologin
Name = vcsa
Password = x
UID = 69
GID = 69
Quota =
Comment =
Gcos = virtual console memory owner
HOME DIR = /dev
Shell = /sbin/nologin
Name = rpc
Password = x
UID = 32
GID = 32
Quota =
Comment =
Gcos = Rpcbind Daemon
HOME DIR = /var/cache/rpcbind
Shell = /sbin/nologin
Name = abrt
Password = x
UID = 173
GID = 173
Quota =
Comment =
Gcos =
HOME DIR = /etc/abrt
Shell = /sbin/nologin
Name = apache
Password = x
UID = 48
GID = 48
Quota =
Comment =
Gcos = Apache
HOME DIR = /var/www
Shell = /sbin/nologin
Name = saslauth
Password = x
UID = 499
GID = 76
Quota =
Comment =
Gcos = "Saslauthd user"
HOME DIR = /var/empty/saslauth
Shell = /sbin/nologin
Name = postfix
Password = x
UID = 89
GID = 89
Quota =
Comment =
Gcos =
HOME DIR = /var/spool/postfix
Shell = /sbin/nologin
Name = qpidd
Password = x
UID = 498
GID = 499
Quota =
Comment =
Gcos = Owner of Qpidd Daemons
HOME DIR = /var/lib/qpidd
Shell = /sbin/nologin
Name = haldaemon
Password = x
UID = 68
GID = 68
Quota =
Comment =
Gcos = HAL daemon
HOME DIR = /
Shell = /sbin/nologin
Name = ntp
Password = x
UID = 38
GID = 38
Quota =
Comment =
Gcos =
HOME DIR = /etc/ntp
Shell = /sbin/nologin
Name = rpcuser
Password = x
UID = 29
GID = 29
Quota =
Comment =
Gcos = RPC Service User
HOME DIR = /var/lib/nfs
Shell = /sbin/nologin
Name = nfsnobody
Password = x
UID = 65534
GID = 65534
Quota =
Comment =
Gcos = Anonymous NFS User
HOME DIR = /var/lib/nfs
Shell = /sbin/nologin
Name = tomcat
Password = x
UID = 91
GID = 91
Quota =
Comment =
Gcos = Apache Tomcat
HOME DIR = /usr/share/tomcat6
Shell = /sbin/nologin
Name = webalizer
Password = x
UID = 67
GID = 67
Quota =
Comment =
Gcos = Webalizer
HOME DIR = /var/www/usage
Shell = /sbin/nologin
Name = sshd
Password = x
UID = 74
GID = 74
Quota =
Comment =
Gcos = Privilege-separated SSH
HOME DIR = /var/empty/sshd
Shell = /sbin/nologin
Name = tcpdump
Password = x
UID = 72
GID = 72
Quota =
Comment =
Gcos =
HOME DIR = /
Shell = /sbin/nologin
Name = oprofile
Password = x
UID = 16
GID = 16
Quota =
Comment =
Gcos = Special user account to be used by OProfile
HOME DIR = /home/oprofile
Shell = /sbin/nologin
Name = amrood
Password = x
UID = 500
GID = 500
Quota =
Comment =
Gcos =
HOME DIR = /home/amrood
Shell = /bin/bash
Name = mailnull
Password = x
UID = 47
GID = 47
Quota =
Comment =
Gcos =
HOME DIR = /var/spool/mqueue
Shell = /sbin/nologin
Name = smmsp
Password = x
UID = 51
GID = 51
Quota =
Comment =
Gcos =
HOME DIR = /var/spool/mqueue
Shell = /sbin/nologin
Name = mysql
Password = x
UID = 27
GID = 27
Quota =
Comment =
Gcos = MySQL Server
HOME DIR = /var/lib/mysql
Shell = /bin/bash
Name = named
Password = x
UID = 25
GID = 25
Quota =
Comment =
Gcos = Named

HOME DIR = /var/named
Shell = /sbin/nologin
Name = qemu
Password = x
UID = 107
GID = 107
Quota =
Comment =
Gcos = qemu user

HOME DIR = /
Shell = /sbin/nologin
Name = com
Password = x
UID = 501
GID = 501
Quota =
Comment =
Gcos =
HOME DIR = /home/com
Shell = /bin/bash
Name = railo
Password = x
UID = 497
GID = 495
Quota =
Comment =
Gcos =
HOME DIR = /opt/railo
Shell = /bin/false
Name = root
Password = x
UID = 0
GID = 0
Quota =
Comment =
Gcos = root
HOME DIR = /root
Shell = /bin/bash
Name = bin
Password = x
UID = 1
GID = 1
Quota =
Comment =
Gcos = bin
HOME DIR = /bin
Shell = /sbin/nologin
Name = daemon
Password = x
UID = 2
GID = 2
Quota =
Comment =
Gcos = daemon
HOME DIR = /sbin
Shell = /sbin/nologin
Name = adm
Password = x
UID = 3
GID = 4
Quota =
Comment =
Gcos = adm
HOME DIR = /var/adm
Shell = /sbin/nologin
Name = lp
Password = x
UID = 4
GID = 7
Quota =
Comment =
Gcos = lp
HOME DIR = /var/spool/lpd
Shell = /sbin/nologin
Name = sync
Password = x
UID = 5
GID = 0
Quota =
Comment =
Gcos = sync
HOME DIR = /sbin
Shell = /bin/sync
Name = shutdown
Password = x
UID = 6
GID = 0
Quota =
Comment =
Gcos = shutdown
HOME DIR = /sbin
Shell = /sbin/shutdown
Name = halt
Password = x
UID = 7
GID = 0
Quota =
Comment =
Gcos = halt
HOME DIR = /sbin
Shell = /sbin/halt
Name = mail
Password = x
UID = 8
GID = 12
Quota =
Comment =
Gcos = mail
HOME DIR = /var/spool/mail
Shell = /sbin/nologin
Name = uucp
Password = x
UID = 10
GID = 14
Quota =
Comment =
Gcos = uucp
HOME DIR = /var/spool/uucp
Shell = /sbin/nologin
Name = operator
Password = x
UID = 11
GID = 0
Quota =
Comment =
Gcos = operator
HOME DIR = /root
Shell = /sbin/nologin
Name = games
Password = x
UID = 12
GID = 100
Quota =
Comment =
Gcos = games
HOME DIR = /usr/games
Shell = /sbin/nologin
Name = gopher
Password = x
UID = 13
GID = 30
Quota =
Comment =
Gcos = gopher
HOME DIR = /var/gopher
Shell = /sbin/nologin
Name = ftp
Password = x
UID = 14
GID = 50
Quota =
Comment =
Gcos = FTP User
HOME DIR = /var/ftp
Shell = /sbin/nologin
Name = nobody
Password = x
UID = 99
GID = 99
Quota =
Comment =
Gcos = Nobody
HOME DIR = /
Shell = /sbin/nologin
Name = dbus
Password = x
UID = 81
GID = 81
Quota =
Comment =
Gcos = System message bus
HOME DIR = /
Shell = /sbin/nologin
Name = vcsa
Password = x
UID = 69
GID = 69
Quota =
Comment =
Gcos = virtual console memory owner
HOME DIR = /dev
Shell = /sbin/nologin
Name = rpc
Password = x
UID = 32
GID = 32
Quota =
Comment =
Gcos = Rpcbind Daemon
HOME DIR = /var/cache/rpcbind
Shell = /sbin/nologin
Name = abrt
Password = x
UID = 173
GID = 173
Quota =
Comment =
Gcos =
HOME DIR = /etc/abrt
Shell = /sbin/nologin
Name = apache
Password = x
UID = 48
GID = 48
Quota =
Comment =
Gcos = Apache
HOME DIR = /var/www
Shell = /sbin/nologin
Name = saslauth
Password = x
UID = 499
GID = 76
Quota =
Comment =
Gcos = "Saslauthd user"
HOME DIR = /var/empty/saslauth
Shell = /sbin/nologin
Name = postfix
Password = x
UID = 89
GID = 89
Quota =
Comment =
Gcos =
HOME DIR = /var/spool/postfix
Shell = /sbin/nologin
Name = qpidd
Password = x
UID = 498
GID = 499
Quota =
Comment =
Gcos = Owner of Qpidd Daemons
HOME DIR = /var/lib/qpidd
Shell = /sbin/nologin
Name = haldaemon
Password = x
UID = 68
GID = 68
Quota =
Comment =
Gcos = HAL daemon
HOME DIR = /
Shell = /sbin/nologin
Name = ntp
Password = x
UID = 38
GID = 38
Quota =
Comment =
Gcos =
HOME DIR = /etc/ntp
Shell = /sbin/nologin
Name = rpcuser
<table>
<thead>
<tr>
<th>Password</th>
<th>UID</th>
<th>GID</th>
<th>Quota</th>
<th>Comment</th>
<th>Gcos</th>
<th>HOME DIR</th>
<th>Shell</th>
<th>Name</th>
<th>Password</th>
<th>UID</th>
<th>GID</th>
<th>Quota</th>
<th>Comment</th>
<th>Gcos</th>
<th>HOME DIR</th>
<th>Shell</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>29</td>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```perl
Password = x
UID = 29
GID = 29
Quota =
Comment =
Gcos = RPC Service User
HOME DIR = /var/lib/nfs
Shell = /sbin/nologin
Name = nfsnobody
Password = x
UID = 65534
GID = 65534
Quota =
Comment =
Gcos = Anonymous NFS User
HOME DIR = /var/lib/nfs
Shell = /sbin/nologin
Name = tomcat
Password = x
UID = 91
GID = 91
Quota =
Comment =
Gcos = Apache Tomcat
HOME DIR = /usr/share/tomcat6
Shell = /sbin/nologin
Name = webalizer
Password = x
UID = 67
GID = 67
Quota =
Comment =
Gcos = Webalizer
```
HOME DIR = /var/www/usage
Shell = /sbin/nologin
Name = sshd
Password = x
UID = 74
GID = 74
Quota =
Comment =
Gcos = Privilege-separated SSH

HOME DIR = /var/empty/sshd
Shell = /sbin/nologin
Name = tcpdump
Password = x
UID = 72
GID = 72
Quota =
Comment =
Gcos =

HOME DIR = /
Shell = /sbin/nologin
Name = oprofile
Password = x
UID = 16
GID = 16
Quota =
Comment =
Gcos = Special user account to be used by OProfile

HOME DIR = /home/oprofile
Shell = /sbin/nologin
Name = amrood
Password = x
UID = 500
GID = 500
Quota =
Comment =
Gcos =
HOME DIR = /home/amrood
Shell = /bin/bash
Name = mailnull
Password = x
UID = 47
GID = 47
Quota =
Comment =
Gcos =
HOME DIR = /var/spool/mqueue
Shell = /sbin/nologin
Name = smmsp
Password = x
UID = 51
GID = 51
Quota =
Comment =
Gcos =
HOME DIR = /var/spool/mqueue
Shell = /sbin/nologin
Name = mysql
Password = x
UID = 27
GID = 27
Quota =
Comment =
Gcos = MySQL Server
HOME DIR = /var/lib/mysql
Shell = /bin/bash
Name = named
Password = x
UID = 25
GID = 25
Quota =
Comment =
Gcos = Named
HOME DIR = /var/named
Shell = /sbin/nologin
Name = qemu
Password = x
UID = 107
GID = 107
Quota =
Comment =
Gcos = qemu user
HOME DIR = /
Shell = /sbin/nologin
Name = com
Password = x
UID = 501
GID = 501
Quota =
Comment =
Gcos =
HOME DIR = /home/com
Shell = /bin/bash
Name = railo
Password = x
UID = 497
GID = 495
Quota =
Comment =
Gcos =
### endservent

#### Description
This function tells the system you no longer expect to read entries from the services file using getservent.

#### Syntax
Following is the simple syntax for this function:

```
endservent
```

#### Return Value
This function does not return any value.

#### Example
Following is the example code showing its basic usage:

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

while($name, $aliases, $port_number, $protocol_name = getservent()){
    print "Name = $name\n";
    print "Aliases = $aliases\n";
    print "Port Number = $port_number\n";
    print "Protocol Name = $protocol_name\n";
}

setservent();   # Rewind the database;

while($name, $aliases, $port_number, $protocol_name = getservent()){
```
When above code is executed, it produces the following result:

```

```

### `eof`

**Description**

This function returns 1 if the next read on `FILEHANDLE` will return end of file, or if `FILEHANDLE` is not open.

An `eof` without an argument uses the last file read. Using `eof()` with empty parentheses is very different. It refers to the pseudo file formed from the files listed on the command line and accessed via the `<>` operator.

**Syntax**

Following is the simple syntax for this function:

```
eof FILEHANDLE

eof()

eof
```

**Return Value**

This function returns `undef` if `FILEHANDLE` is not at end of file and 1 if `FILEHANDLE` will report end of file on next read.

**Example**

Following is the example code showing its basic usage:
# insert dashes just before last line of last file
while (<>)
{
  if (eof()) {
    # check for end of last file
    print "----------
";
  }
  print;
  last if eof(); # needed if we're reading from a terminal
}

When above code is executed, it produces the following result:

eval

Description
This function evaluates EXPR at execution time as if EXPR were a separate Perl script. This allows you to use a separate, perhaps user-supplied, piece of Perl script within your program. An eval EXPR statement is evaluated separately each time the function is called.

The second form evaluates BLOCK when the rest of the script is parsed (before execution).

Syntax
Following is the simple syntax for this function:

eval EXPR

eval BLOCK

Return Value
This function returns value of last evaluated statement in EXPR or BLOCK

Example
Following is the example code showing its basic usage:
When above code is executed, it produces the following result:

```
```

**exec**

**Description**
This function executes a system command (directly, not within a shell) and never returns to the calling script, except if the command specified does not exist and has been called directly, instead of indirectly through a shell. The operation works as follows:

If there is only one scalar argument that contains no shell metacharacters, then the argument is converted into a list and the command is executed directly, without a shell.

If there is only one scalar argument that contains shell metacharacters, then the argument is executed through the standard shell, usually /bin/sh on Unix.

If LIST is more than one argument, or an array with more than one value, then the command is executed directly without the use of a shell.

**Syntax**
Following is the simple syntax for this function:

```
exec EXPR LIST
```

```
exec LIST
```

**Return Value**
This function returns 0 only if the command specified cannot be executed.

**Example**
Following is the example code showing its basic usage:

```
```

When above code is executed, it produces the following result:

```
```
exists

Description
This function returns true if the specified hash or array key exists, regardless of the corresponding value, even if it's undef. If EXPR is a subroutine, then exists will return 1 if the subroutine has been declared (but not necessarily defined), 0 if not.

If LIST is more than one argument, or an array with more than one value, then the command is executed directly without the use of a shell.

Syntax
Following is the simple syntax for this function:

```
exists EXPR
```

Return Value
This function returns 0 if hash element or array index does not exist, or if the subroutine has not been declared and 1 if hash element or array index does exist, or if the subroutine has not been declared.

Example
Following is the example code showing its basic usage:

```
```

When above code is executed, it produces the following result:

```
```

exit

Description
This function evaluates EXPR, exits the Perl interpreter, and returns the value as the exit value. Always runs all END{} blocks defined in the script (and imported packages) before exiting. If EXPR is omitted, then the interpreter exits with a value of 0. Should not be used to exit from a subroutine; either use eval and die or use return.
**Syntax**
Following is the simple syntax for this function:

```
exit EXPR
```

```
exit
```

**Return Value**
This function does not return any value.

**Example**
Following is the example code showing its basic usage:

When above code is executed, it produces the following result:

```
exp
```

**Description**
This function returns e (the natural logarithm base) to the power of EXPR. If EXPR is omitted, gives exp($_).

**Syntax**
Following is the simple syntax for this function:

```
exp EXPR
```

```
exp
```

**Return Value**
This function does not return any value.

**Example**
Following is the example code showing its basic usage:
When above code is executed, it produces the following result:

---

**fcntl**

**Description**

This function is the Perl version of the system `fcntl()` function. Performs the function specified by `FUNCTION`, using `SCALAR` on `FILEHANDLE`. `SCALAR` either contains a value to be used by the function or is the location of any returned information.

**Syntax**

Following is the simple syntax for this function:

```
fcntl FILEHANDLE, FUNCTION, SCALAR
```

**Return Value**

This function returns 0 but true if the return value from the `fcntl()` is 0 Value returned by system and undef on failure.

**Example**

Following is the example code showing its basic usage:

When above code is executed, it produces the following result:

---

**fileno**

**Description**

This function returns the file descriptor number (as used by C and POSIX functions) of the specified `FILEHANDLE`. This is generally useful only for using the `select` function and any low-level tty functions.
Syntax
Following is the simple syntax for this function:

```
fileno FILEHANDLE
```

Return Value
This function returns File descriptor (numeric) of FILEHANDLE and undef on failure.

Example
When above code is executed, it produces the following result:

```

```

flock

Description
This function supports file locking on the specified FILEHANDLE using the system flock(), fcntl() locking, or lockf(). The exact implementation used is dependent on what your system supports. OPERATION is one of the static values defined here.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCK_SH</td>
<td>Set shared lock.</td>
</tr>
<tr>
<td>LOCK_EX</td>
<td>Set exclusive lock.</td>
</tr>
<tr>
<td>LOCK_UN</td>
<td>Unlock specified file.</td>
</tr>
<tr>
<td>LONG_NB</td>
<td>Set lock without blocking.</td>
</tr>
</tbody>
</table>

Syntax
Following is the simple syntax for this function:

```
flock FILEHANDLE, OPERATION
```
Return Value
This function returns 0 on failure to set/unset lock and 1 on success to set/unset lock.

Example
Following is the example code showing its basic usage:

Return Value
This function returns undef on failure to fork and Child process ID to parent on success 0 to child on success.

Example
Following is the example code showing its basic usage:

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

$pid = fork();
if( $pid == 0 ){
    print "This is child process\n";
    print "Child process is existing\n";
    exit 0;
}
```
When above code is executed, it produces the following result:

This is parent process and child ID is 18641
Parent process is existing
This is child process
Child process is existing

format

As stated earlier that Perl stands for Practical Extraction and Reporting Language, and we'll now discuss using Perl to write reports.

Perl uses a writing template called a 'format' to output reports. To use the format feature of Perl, you must:

- Define a Format
- Pass the data that will be displayed on the format
- Invoke the Format

Define a Format

Following is the syntax to define a Perl format

```perl
format FormatName =
  fieldline
  value_one, value_two, value_three
  fieldline
  value_one, value_two
.
```

**FormatName** represents the name of the format. The fieldline is the specific way the data should be formatted. The values lines represent the values that will be entered into the field line. You end the format with a single period.

**fieldline** can contain any text or fieldholders. Fieldholders hold space for data that will be placed there at a later date. A fieldholder has the format:
This fieldholder is left-justified, with a field space of 5. You must count the @ sign and the < signs to know the number of spaces in the field. Other field holders include:

- @>>>> right-justified
- @|||| centered
- @####.## numeric field holder
- @* multiline field holder

An example format would be:

```perl
format EMPLOYEE =
  ==================================================
  @<<<<<<<<<<<<<<<<<<<<<< @<<
  $name $age
  @####.##
  $salary
  ==================================================
```

In this example $name would be written as left justify within 22 character spaces and after that age will be written in two spaces.

**Invoke the Format to write Data**

In order to invoke this format declaration we would use the write keyword:

```perl
write EMPLOYEE; #send to the output
```

The problem is that the format name is usually the name of an open file handle, and the write statement will send the output to this file handle. As we want the data sent to the STDOUT, we must associate EMPLOYEE with the STDOUT filehandle. First, however, we must make sure that that STDOUT is our selected file handle, using the select() function

```perl
select(STDOUT);
```

We would then associate EMPLOYEE with STDOUT by setting the new format name with STDOUT, using the special variable $~
When we now do a write(), the data would be sent to STDOUT. Remember: if you didn't have STDOUT set as your default file handle, you could revert back to the original file handle by assigning the return value of select to a scalar value, and using select along with this scalar variable after the special variable is assigned the format name, to be associated with STDOUT.

The above example will generate a report in the following format

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kirsten</td>
<td>12</td>
</tr>
<tr>
<td>Mohammad</td>
<td>35</td>
</tr>
<tr>
<td>Suhi</td>
<td>15</td>
</tr>
<tr>
<td>Namrat</td>
<td>10</td>
</tr>
</tbody>
</table>

**Defining a Report Header**

Everything looks fine. But you would be interested in adding a header to your report. This header will be printed on top of each page. It is very simple to do this. Apart from defining a template you would have to define a header which will have same name but appended with _TOP keyword as follows

```
format EMPLOYEE_TOP =
     ------------------------
     | Name       Age       |
     ------------------------
     .
```

Now your report will look like

```
------------------------
| Name       Age       |
------------------------
Kirsten      12         
Mohammad     35         
Suhi         15         
Namrat       10         
```
Defining a Pagination & Number of Lines on a Page

What about if your report is taking more than one page? You have a solution for that. Use \$\% variable along with header as follows:

```perl
format EMPLOYEE_TOP =
  ------------------------
  Name    Age    Page @<
  ------------------------
  \$\% .
```

Now your output will look like:

```
----------------------------
Name    Age    Page 1
----------------------------
Kirsten  12
Mohammad 35
Suhi     15
Namrat   10
```

You can set the number of lines per page using special variable \$= (or \$FORMAT_LINES_PER_PAGE) By default \$= will be 60

Defining a Report Footer

One final thing is left which is footer. Very similar to header, you can define a footer and it will be written after each page. Here you will use _BOTTOM keyword instead of _TOP.

```perl
format EMPLOYEE_BOTTOM =
  End of Page @<
  \$\%
  .
```

This will give you following result:

```
----------------------------
Name    Age    Page 1
----------------------------
Kirsten  12
```
For a complete set of variables related to formatting, please refer to Perl Special Variables section.

formline

Description
This function is used by the format function and related operators. It formats LIST according to the contents of PICTURE into the output accumulator variable $^A. The value is written out to a filehandle when a write is done.

Syntax
Following is the simple syntax for this function:

formline PICTURE, LIST

Return Value
This function always returns 1.

Example
Following is the example code showing its basic usage:

When above code is executed, it produces the following result:

getc

Description
This function reads the next character from FILEHANDLE (or STDIN if none specified), returning the value.
Syntax
Following is the simple syntax for this function:

```
getc FILEHANDLE
```

Return Value
This function returns undef on error or end of file and value of character read from FILEHANDLE.

Example
Following is the example code showing its basic usage:

```
#!/usr/bin/perl

$key = getc(STDIN);
print "Entered value is $key\n";
```

When above code is executed, it produces the following result:

```
Entered value is 4
```

getgrent

Description
This function iterates over the entries in the /etc/group file. Returns the following in a list context:

```
($name, $passwd, $gid, $members)
```

The $members scalar contains a space-separated list of the login names that are members of the group. Returns the group name only when used in a scalar context. Under Windows, consider using the Win32API::Net module.

Syntax
Following is the simple syntax for this function:

```
getgrent
```
Return Value

This function returns Group name in scalar context and in list context (Name, Password, Group ID, and member list.

Example

Following is the example code showing its basic usage:

```
#!/usr/bin/perl

while( ($name,$passwd,$gid,$members) = getgrent() ){
    print "Name  = $name\n";
    print "Password  = $passwd\n";
    print "GID  = $gid\n";
    print "Members  = $members\n";
}
```

When above code is executed, it produces the following result:

```
Name  = root
Password  = x
GID  = 0
Members  = root
Name  = bin
Password  = x
GID  = 1
Members  = root bin daemon
Name  = daemon
Password  = x
GID  = 2
Members  = root bin daemon
Name  = sys
Password  = x
GID  = 3
Members  = root bin adm
Name  = adm
Password  = x
```
GID = 4
Members = root adm daemon
Name = tty
Password = x

GID = 5
Members =
.
.
.
Name = fuse
Password = x

GID = 496
Members =
Name = kvm
Password = x

GID = 36
Members = qemu
Name = qemu
Password = x

GID = 107
Members =
Name = com
Password = x

GID = 501
Members =
Name = webgrp
Password = x

GID = 502
Members = com
Name = railo
Password = x

GID = 495
Members =
getgrgid

Description
This function looks up the group file entry by group ID. Returns the following in a list context:

($name, $passwd, $gid, $members)

The $members scalar contains a space-separated list of the login names that are members of the group. Returns the group name in a scalar context. For a more efficient method of retrieving the entire groups file, see getgrent. Under Windows, consider using the Win32API::Net module.

Syntax
Following is the simple syntax for this function:

getgrgid GID

Return Value
This function returns in scalar context it returns Group name and in list context (Name, Password, Group ID, and member list.

Example
Following is the example code showing its basic usage:

```
#!/usr/bin/perl

while( ($name,$passwd,$gid,$members) = getgrent() ){
    ($name,$passwd,$gid,$members) = getgrgid $gid;
    print "Name   = $name\n";
    print "Password = $passwd\n";
    print "GID     = $gid\n";
    print "Members = $members\n";
}
```

When above code is executed, it produces the following result:

```
Name   = root
Password = x
GID     = 0
```
Members = root
Name = root
Password = x
GID = 0
Members = root
Name = root
Password = x
GID = 0
Members = root
Name = root
Password = x
GID = 0
Members = root
Name = root
Password = x
GID = 0
Members = root
Name = root
Password = x
GID = 0
Members = root
Name = root
Password = x
GID = 0
Members = root
Name = root
Password = x
GID = 0
Members = root
Name = root
Password = x
GID = 0
Members = root
Name = root
Password = x
GID = 0
Members = root
Name = root
getgrnam

Description
This function looks up the group file entry by group name. Returns the following in a list context: ($name, $passwd, $gid, $members)

The $members scalar contains a space-separated list of the login names that are members of the group. Returns the group name in a scalar context. For a more efficient method of retrieving the entire groups file, see getgrent. Under Windows, consider using the Win32API::Net module.

Syntax
Following is the simple syntax for this function:

getgrnam NAME

Return Value
This function returns Group name in scalar context and Name, Password, Group ID, and member list in list context.

Example
Following is the example code showing its basic usage:

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

while( ($name, $passwd, $gid, $members) = getgrent() ){
    ($name, $passwd, $gid, $members) = getgrnam $name;
    print "Name  = $name\n";
    print "Password  = $passwd\n";
}
```
When above code is executed, it produces the following result:

Name  = root
Password  = x
GID  = 0
Members  = root
Name  = root
Password  = x
GID  = 0
Members  = root
Name  = root
Password  = x
GID  = 0
Members  = root
Name  = root
Password  = x
GID  = 0
Members  = root
Name  = root
**gethostbyaddr**

**Description**
This function contacts the system's name-resolving service, returning a list of information for the host ADDR of type ADDRTYPE, as follows: ($name, $aliases, $addrtype, $length, @addrs)

The @addrs array contains a list of packed binary addresses. In a scalar context, returns the host address.

**Syntax**
Following is the simple syntax for this function:

```
gethostbyaddr ADDR, ADDRTYPE
```

**Return Value**
This function returns undef on error and otherwise host name in scalar context and empty list on error otherwise host record in list context.

**Example**
Following is the example code showing its basic usage:

```
#!/usr/bin/perl
use Socket;

$iaddr = inet_aton("127.1"); # or whatever address
$name = gethostbyaddr($iaddr, AF_INET);
print "Host name is $name\n";
```

When above code is executed, it produces the following result

```
Host name is ip-50-62-147-141.ip.secureserver.net
```

**gethostbyname**

**Description**
This function contacts the system's name-resolving service, returning a list of information for the host ADDR of type ADDRTYPE, as follows: ($name, $aliases, $addrtype, $length, @addrs)
The @addrs array contains a list of packed binary addresses. In a scalar context, returns the host address.

**Syntax**

Following is the simple syntax for this function:

```
gethostbyname NAME
```

**Return Value**

This function returns undef on error and otherwise host name in scalar context and empty list on error otherwise host record in list context.

**Example**

Following is the example code showing its basic usage:

```
#!/usr/bin/perl
use Socket;

($name, $aliases, $addrtype, $length, @addrs) = gethostbyname "amrood.com";
print "Host name is $name\n";
print "Aliases is $aliases\n";
```

When above code is executed, it produces the following result

```
Host name is amrood.com
Aliases is
```

**gethostent**

**Description**

This function iterates over the entries in the host file. It returns the following in a list context: ($name, $aliases, $addrtype, $length, @addrs)

**Syntax**

Following is the simple syntax for this function:

```
gethostent
```
Return Value

This function returns undef on error and otherwise host name in scalar context and empty list on error otherwise host record(name, aliases, address type, length, list of addresses) in list context.

Example

Following is the example code showing its basic usage:

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

while( ($name, $aliases, $addrtype, $length, @addrs) = gethostent() ) {
    print "Name  = $name\n";
    print "Aliases  = $aliases\n";
    print "Addr Type  = $addrtype\n";
    print "Length  = $length\n";
    print "Addrs  = @addrs\n";
}
```

When above code is executed, it produces the following result

```
Name  = ip-50-62-147-141.ip.secureserver.net
Aliases  = ip-50-62-147-141 localhost.secureserver.net
          localhost.localdomain localhost
Addr Type  = 2
Length  = 4
Addrs  =
```

getlogin

Description

This function returns the user's name, as discovered by the system function getlogin(). Under Windows, use the Win32::LoginName() function instead.

Syntax

Following is the simple syntax for this function:

```perl
getlogin
```
Return Value
This function returns undef on failure and user’s login name on success.

Example
Following is the example code showing its basic usage:

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

$login = getlogin || getpwuid($<) || "TutorialsPoint";

print "Login ID is $login\n";
```

When above code is executed, it produces the following result

Login ID is apache

getnetbyaddr

Description
This function returns the information for the network specified by ADDR and type ADDRTYPE in a list context: ($name, $aliases, $addrtype, $net)

Syntax
Following is the simple syntax for this function:

```perl
getnetbyaddr ADDR, ADDRTYPE
```

Return Value
This function returns undef on error otherwise Network address in scalar context and empty list on error otherwise Network record (name, aliases, address type, network address) in list context.

Example
Following is the example code showing its basic usage:

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

use Socket;
```
$iaddr = inet_aton("127.1"); # or whatever address
($name, $aliases, $addrtype, $net) = getnetbyaddr($iaddr, AF_INET);

print "Name = $name\n";
print "Aliases = $aliases\n";
print "Addrtype = $addrtype\n";
print "Net = $net\n";

When above code is executed, it produces the following result

Name = default
Aliases =
Addrtype = 2
Net = 0

getnetbyname

Description
This function returns the information for the network specified by NAME(in list context) ($name, $aliases, $addrtype, $net)

Syntax
Following is the simple syntax for this function:

getnetbyname NAME

Return Value
This function returns undef on error otherwise Network address in scalar context and empty list on error otherwise Network record (name, aliases, address type, network address) in list context.

Example
Following is the example code showing its basic usage:

#!/usr/bin/perl

use Socket;
Perl

```perl
($name, $aliases, $addrtype, $net) = getnetbyname("loopback");

print "Name = $name\n";
print "Aliases = $aliases\n";
print "Addrtype = $addrtype\n";
print "Net = $net\n";
```

When above code is executed, it produces the following result

```
Name = loopback
Aliases =
Addrtype = 2
Net = 2130706432
```

**getnetent**

**Description**
This function gets the next entry from the /etc/networks file, returning: ($name, $aliases, $addrtype, $net)

If /etc/networks file is empty then it would not return anything and call will fail..

**Syntax**
Following is the simple syntax for this function:

```
getnetent
```

**Return Value**
This function returns undef on error otherwise Network address in scalar context and empty list on error otherwise Network record (name, aliases, address type, network address) in list context.

**Example**
Following is the example code showing its basic usage:

```
#!/usr/bin/perl
```
use Socket;

while ( ($name, $aliases, $addrtype, $net) = getnetent() ){
    print "Name = $name\n";
    print "Aliases = $aliases\n";
    print "Addrtype = $addrtype\n";
    print "Net = $net\n";
}

When above code is executed, it produces the following result:

Name = default
Aliases =
Addrtype = 2
Net = 0
Name = loopback
Aliases =
Addrtype = 2
Net = 2130706432
Name = link-local
Aliases =
Addrtype = 2
Net = 2851995648

getpeername

Description
This function returns the packed socket address of the remote host attached to SOCKET.

Syntax
Following is the simple syntax for this function:

getpeername SOCKET
**Return Value**

This function returns undef on error otherwise packed socket address in scalar context.

**Example**

Following is the example code showing its basic usage, here SOCK is the socket ID of the peer socket:

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

use Socket;

$hersockaddr = getpeername(SOCK);
($port, $iaddr) = sockaddr_in($hersockaddr);
$herhostname = gethostbyaddr($iaddr, AF_INET);
$herstraddr  = inet_ntoa($iaddr);
```

When above code is executed, it produces the following result

**getpgid**

**Description**

This function returns the process group for the process ID specified by EXPR, or the current process group if none is specified.

**Syntax**

Following is the simple syntax for this function:

```perl
getpgid EXPR
getpgid
```

**Return Value**

This function returns process group ID.
Following is the example code showing its basic usage:

```
#!/usr/bin/perl

$pgid = getpgrp();

print "Current process Group ID $pgid\n";
```

When above code is executed, it produces the following result

```
Current process Group ID 12055
```

**getppid**

**Description**

This function returns the process ID of the parent process.

**Syntax**

Following is the simple syntax for this function:

```
getppid
```

**Return Value**

This function returns process ID of the parent process.

**Example**

Following is the example code showing its basic usage:

```
#!/usr/bin/perl

$ppid = getppid();

print "Parent Process ID $ppid\n";
```

When above code is executed, it produces the following result

```
Parent Process ID 3830
```
getpriority

Description
This function returns the current priority for a process (PRIO_PROCESS), process group (PRIO_PGRP) or user (PRIO_USER).

The argument WHICH specifies what entity to set the priority for one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER, and WHO is the process ID or user ID to set. A value of 0 for WHO defines the current process, process group, or user. This produces a fatal error on systems that don't support the system getpriority( ) function.

Syntax
Following is the simple syntax for this function:

getpriority WHICH, WHO

Return Value
This function returns undef on error otherwise current priority.

Example
Following is the example code showing its basic usage:

#!/usr/bin/perl

When above code is executed, it produces the following result

getprotobynumber

Description
This function translates the protocol NAME into its corresponding number in a scalar context, and its number and associated information in a list context:($name, $aliases, $protocol_number)

Syntax
Following is the simple syntax for this function:

getprotobynumber NAME
**Return Value**

This function returns undef on error otherwise protocol number in scalar context and empty list on error protocol record (name, aliases, protocol number) in list context.

**Example**

Following is the example code showing its basic usage:

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

($name, $aliases, $protocol_number) = getprotobynumber("tcp");

print "Name = $name\n";
p
print "Aliases = $aliases\n";
p
print "Protocol Number = $protocol_number\n";
```

When above code is executed, it produces the following result

```
Name = tcp
Aliases = TCP
Protocol Number = 6
```

**getprotobynumber**

**Description**

This function translates the protocol NUMBER into its corresponding name in a scalar context, and its name and associated information in a list context:($name, $aliases, $protocol_number).

**Syntax**

Following is the simple syntax for this function:

```
getprotobynumber NUMBER
```

**Return Value**

This function returns undef on error otherwise protocol number in scalar context and empty list on error protocol record (name, aliases, protocol number) in list context.
**Example**
Following is the example code showing its basic usage:

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

($name, $aliases, $protocol_number) = getprotobynumber(6);

print "Name = $name\n";
print "Aliases = $aliases\n";
print "Protocol Number = $protocol_number\n";
```

When above code is executed, it produces the following result:

```
Name = tcp
Aliases = TCP
Protocol Number = 6
```

**getprotoent**

**Description**
This function returns the next entry from the list of valid protocols: ($name, $aliases, $protocol_number)

**Syntax**
Following is the simple syntax for this function:

```
getprotoent
```

**Return Value**
This function returns undef on error otherwise protocol number in scalar context and empty list on error protocol record (name, aliases, protocol number) in list context.

**Example**
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl
```
while(($name, $aliases, $protocol_number) = getprotoent()){
    print "Name = $name\n";
    print "Aliases = $aliases\n";
    print "Protocol Number = $protocol_number\n";
}

When above code is executed, it produces the following result

Name = ip
Aliases = IP
Protocol Number = 0
Name = hopopt
Aliases = HOPOPT
Protocol Number = 0
Name = icmp
Aliases = ICMP
Protocol Number = 1
Name = igmp
Aliases = IGMP
Protocol Number = 2
Name = ggp
Aliases = GGP
Protocol Number = 3
Name = ipencap
Aliases = IP-ENCAP
Protocol Number = 4
.
.
.
Name = udplite
Aliases = UDPLite
Protocol Number = 136
Name = mpls-in-ip
Aliases = MPLS-in-IP
Protocol Number = 137
getpwent

Description
This function returns the next password entry from the /etc/passwd file. This is used in combination with the setpwent and endpwent functions to iterate over the password file. In a list context, returns

($name, $passwd, $uid, $gid, $quota, $comment, $gcos, $dir, $shell) = getpwent;

Syntax
Following is the simple syntax for this function:

getpwent

Return Value
This function returns username in scalar context and user record (name, password, user ID, group ID, quote, comment, real name, home directory, shell) in list context.

Example
Following is the example code showing its basic usage:

#!/usr/bin/perl

while(($name, $passwd, $uid, $gid, $quota, $comment, $gcos, $dir, $shell) = getpwent()){

}
print "Name = $name\n";
print "Password = $passwd\n";
print "UID = $uid\n";
print "GID = $gid\n";
print "Quota = $quota\n";
print "Comment = $comment\n";
print "Gcos = $gcos\n";
print "HOME DIR = $dir\n";
print "Shell = $shell\n";
}

When above code is executed, it produces the following result

Name = root
Password = x
UID = 0
GID = 0
Quota =
Comment =
Gcos = root
HOME DIR = /root
Shell = /bin/bash
Name = bin
Password = x
UID = 1
GID = 1
Quota =
Comment =
Gcos = bin
HOME DIR = /bin
Shell = /sbin/nologin
.
.
.
Name = com
Password = x
UID = 501
GID = 501
Quota =
Comment =
Gcos =
HOME DIR = /home/com
Shell = /bin/bash
Name = railo
Password = x
UID = 497
GID = 495
Quota =
Comment =
Gcos =
HOME DIR = /opt/railo

gotpwnam

Description
This function returns a list of fields In list context, as extracted from the /etc/passwd file, based on the user name specified by EXPR. It's generally used like this:

($name, $passwd, $uid, $gid, $quota, $comment, $gcos, $dir, $shell) = getpwnam ($user);

In a scalar context, returns the numeric user ID. If you are trying to access the whole /etc/passwd file, you should use the getpwent function. If you want to access the details by user ID, use getpwuid.

Syntax
Following is the simple syntax for this function:

gotpwnam EXPR
Return Value

This function returns user ID in scalar context and user record (name, password, user ID, group ID, quote, comment, real name, home directory, shell) in list context.

Example

Following is the example code showing its basic usage:

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

($name, $passwd, $uid, $gid, $quota, $comment, $gcos, $dir, $shell) = getpwnam("root");
print "Name = $name\n";
print "Password = $passwd\n";
print "UID = $uid\n";
print "GID = $gid\n";
print "Quota = $quota\n";
print "Comment = $comment\n";
print "Gcos = $gcos\n";
print "HOME DIR = $dir\n";
print "Shell = $shell\n";
```

When above code is executed, it produces the following result

Name = root
Password = x
UID = 0
GID = 0
Quota =
Comment =
Gcos = root
HOME DIR = /root
Shell = /bin/bash
### getpwuid

#### Description

This function returns a list of fields in list context, as extracted from the /etc/passwd file, based on the user name specified by EXPR. It's generally used like this:

```perl
($name, $passwd, $uid, $gid, $quota, $comment, $gcos, $dir, $shell) = getpwuid ($uid);
```

In a scalar context, returns the user name. If you are trying to access the whole /etc/passwd file, you should use the getpwent function. If you want to access the details by user name, use getpwnam.

#### Syntax

Following is the simple syntax for this function:

```perl
getpwuid EXPR
```

#### Return Value

This function returns user Name in scalar context and user record (name, password, user ID, group ID, quote, comment, real name, home directory, shell) in list context.

#### Example

Following is the example code showing its basic usage:

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

($name, $passwd, $uid, $gid, $quota,  
 $comment, $gcos, $dir, $shell) = getpwuid(0);
print "Name = $name\n";
print "Password = $passwd\n";
print "UID = $uid\n";
print "GID = $gid\n";
print "Quota = $quota\n";
print "Comment = $comment\n";
print "Gcos = $gcos\n";
print "HOME DIR = $dir\n";
```
print "Shell = $shell\n";

When above code is executed, it produces the following result

Name = root
Password = x
UID = 0
GID = 0
Quota =
Comment =
Gcos = root
HOME DIR = /root
Shell = /bin/bash

getservbyname

Description
This function Translates the service NAME for the protocol PROTO, returning the service number in a scalar context and the number and associated information in a list context:

($name, $aliases, $port_number, $protocol_name)

This call returns these values based on /etc/services file.

Syntax
Following is the simple syntax for this function:

getservbyname NAME, PROTO

Return Value
This function returns undef on error otherwise service number in scalar context and empty list on error otherwise Service record (name, aliases, port number, protocol name) in list context.

Example
Following is the example code showing its basic usage:

#!/usr/bin/perl
($name, $aliases, $port_number, $protocol_name) = getservbyname("ftp", "tcp");
print "Name = $name\n";
print "Aliases = $aliases\n";
print "Port Number = $port_number\n";
print "Protocol Name = $protocol_name\n";

When above code is executed, it produces the following result

Name = ftp
Aliases =
Port Number = 21
Protocol Name = tcp

getservbyport

Description
This function Translates the service number PORT for the protocol PROTO, returning the service name in a scalar context and the name and associated information in a list context:

($name, $aliases, $port_number, $protocol_name)

This call returns these values based on /etc/services file.

Syntax
Following is the simple syntax for this function:

getservbyport PORT, PROTO

Return Value
This function returns undef on error otherwise service number in scalar context and empty list on error otherwise Service record (name, aliases, port number, protocol name) in list context.

Example
Following is the example code showing its basic usage:
#!/usr/bin/perl

($name, $aliases, $port_number, $protocol_name) = getservbyport(21, "tcp");

print "Name = $name\n";
print "Aliases = $aliases\n";
print "Port Number = $port_number\n";
print "Protocol Name = $protocol_name\n";

When above code is executed, it produces the following result

Name = ftp
Aliases =
Port Number = 21
Protocol Name = tcp

getservent

Description
This function gets the next entry from the list of service entries, returning:
($name, $aliases, $port_number, $protocol_name)
This call iterate through /etc/services file.

Syntax
Following is the simple syntax for this function:

getservent

Return Value
This function returns undef on error otherwise service name in scalar context and empty list on error otherwise Service record (name, aliases, port number, protocol name) in list context.

Example
Following is the example code showing its basic usage:
#!/usr/bin/perl

while((($name, $aliases, $port_number, $protocol_name) = getservent())){

    print "Name = $name\n";
    print "Aliases = $aliases\n";
    print "Port Number = $port_number\n";
    print "Protocol Name = $protocol_name\n";
}

When above code is executed, it produces the following result

Name = tcpmux
Aliases =
Port Number = 1
Protocol Name = tcp
Name = tcpmux
Aliases =
Port Number = 1
Protocol Name = udp
Name = rje
Aliases =
Port Number = 5
Protocol Name = tcp
Name = rje
Aliases =
Port Number = 5
Protocol Name = udp
getsockoptname

Description
This function returns a packed address of the local end of the network socket SOCKET.

Syntax
Following is the simple syntax for this function:

```perl
getsockname SOCKET
```

Return Value
This function returns undef on error otherwise Packed address of local socket in scalar context.

Example
Following is the example code showing its basic usage:

```perl

```

When above code is executed, it produces the following result
### Description

This function gets the socket options set on SOCKET at the socket implementation level LEVEL for the option OPTNAME. Some sample values for OPTNAME at a socket level are given in Table below:

<table>
<thead>
<tr>
<th>OPTNAME</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SO_DEBUG</td>
<td>Get status of recording of debugging information</td>
</tr>
<tr>
<td>SO_REUSEADDR</td>
<td>Get status of local address reuse</td>
</tr>
<tr>
<td>SO_KEEPALIVE</td>
<td>Get status of keep connections alive</td>
</tr>
<tr>
<td>SO_DONTRoute</td>
<td>Get status of routing bypass for outgoing messages</td>
</tr>
<tr>
<td>SO_LINGER</td>
<td>Get status of linger on close if data is present</td>
</tr>
<tr>
<td>SO_BROADCAST</td>
<td>Get status of permission to transmit broadcast messages</td>
</tr>
<tr>
<td>SO_OOBINLINE</td>
<td>Get status of out-of-band data in band</td>
</tr>
<tr>
<td>SO_SNDBUF</td>
<td>Get buffer size for output</td>
</tr>
<tr>
<td>SO_RCVBUF</td>
<td>Get buffer size for input</td>
</tr>
<tr>
<td>SO_TYPE</td>
<td>Get the type of the socket</td>
</tr>
<tr>
<td>SO_ERROR</td>
<td>Get and clear error on the socket</td>
</tr>
<tr>
<td>TCP_NODELAY</td>
<td>To disable the Nagle buffering algorithm.</td>
</tr>
</tbody>
</table>

What exactly is in the packed string depends in the LEVEL and OPTNAME, consult your system documentation for details.

### Syntax

Following is the simple syntax for this function:

```
getsockopt SOCKET, LEVEL, OPTNAME
```

### Return Value

This function returns undef on error otherwise option value in scalar context.

### Example
Following is the example code showing its basic usage, this will check if Nagle's algorithm is turned on on a socket. But, here you would have to open one socket to provide socked ID in this example:

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

use Socket qw(:all);

defined(my $tcp = getprotobynumber("tcp"))
    or die "Could not determine the protocol number for tcp";
# my $tcp = IPPROTO_TCP; # Alternative

my $packed = getsockopt($socket, $tcp, TCP_NODELAY)
    or die "Could not query TCP_NODELAY socket option: $!";
my $nodelay = unpack("I", $packed);

print "Nagle's algorithm is turned ", $nodelay ? "off\n" : "on\n";
```

When above code is executed, it produces the following result

```
glob

Description
This function returns a list of files matching EXPR as they would be expanded by the standard Bourne shell. If the EXPR does not specify a path, uses the current directory. If EXPR is omitted, the value of $_ is used.

From Perl 5.6 on, expansion is done internally, rather than using an external script. Expansion follows the csh (and any derivatives, including tcsh and bash) style of expansion, which translates as the following:

- Files beginning with a single period are ignored unless EXPR explicitly matches.
- The * character matches zero or more characters of any type.
- The ? character matches one character of any type.
- The [...] construct matches the characters listed, including ranges, as per regular expressions.
```
• The `~` characters matches the home directory; `~name` matches the home directory for the user name.
• The `{..}` construct matches against any of the comma-separated words enclosed in the braces.

**Syntax**
Following is the simple syntax for this function:

```perl
glob EXPR

glob
```

**Return Value**
This function returns `undef` on error otherwise First file in the list of expanded names in scalar context and Empty list on error otherwise List of expanded file names in list context.

**Example**
Following is the example code showing its basic usage:

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

(@file_list) = glob "perl_g*";

print "Returned list of file @file_list\n";
```

When above code is executed, it produces the following result:

```
Returned list of file
```

**gmtime**

**Description**
This function returns a list of values corresponding to the date and time as specified by `EXPR`, or date and time returned by the `time` function if `EXPR` is omitted, localized for the standard Greenwich mean time. The values returned are as follows:

```perl
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = gmtime(time);
```
All list elements are numeric, and come straight out of the C `struct tm'. $sec, $min, and $hour are the seconds, minutes, and hours of the specified time. $mday is the day of the month, and $mon is the month itself, in the range 0..11 with 0 indicating January and 11 indicating December. $year is the number of years since 1900. That is, $year is 123 in year 2023. $wday is the day of the week, with 0 indicating Sunday and 3 indicating Wednesday. $yday is the day of the year, in the range 0..364 (or 0..365 in leap years). $isdst is always 0.

**Syntax**

Following is the simple syntax for this function:

```
gmtime EXPR
```  
gmtime

**Return Value**

This function returns a string of the form: Thu Sep 21 14:52:52 2000 in scalar context and in list context the individual time component values (seconds, minutes, hours, day of month, month, year, day of week, day of year, daylight savings time).

**Example**

Following is the example code showing its basic usage:

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

@weekday = ("Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat");

$local_time = gmtime();

print "Local time = $local_time\n";
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = gmtime(time);
$year = $year + 1900;
print "Formated time = $mday/$mon/$year $hour:$min:$sec $weekday[$wday]\n";
```

When above code is executed, it produces the following result:

```
Local time = Sun Sep  1 09:06:41 2013
```
goto

Description
This function has three forms, the first form causes the current execution point to jump to the point referred to as LABEL. A goto in this form cannot be used to jump into a loop or external function. You can only jump to a point within the same scope.

The second form expects EXPR to evaluate to a recognizable LABEL. In general, you should be able to use a normal conditional statement or function to control the execution of a program, so its use is deprecated.

The third form substitutes a call to the named subroutine for the currently running subroutine. The new subroutine inherits the argument stack and other features of the original subroutine; it becomes impossible for the new subroutine even to know that it was called by another name.

Syntax
Following is the simple syntax for this function:

```
goto LABEL

goto EXPR

goto &NAME
```

Return Value
This function does not return any value.

Example
Following is the example code showing its basic usage:

```
#!/usr/bin/perl

$count = 0;

START:
```
$count = $count + 1;

if( $count > 4 ){
    print "Exiting program\n";
}else{
    print "Count = $count, Jumping to START:\n";
    goto START;
}

When above code is executed, it produces the following result:

Count = 1, Jumping to START:
Count = 2, Jumping to START:
Count = 3, Jumping to START:
Count = 4, Jumping to START:
Exiting program

grep

Description
This function extract any elements from LIST for which EXPR is TRUE.

Syntax
Following is the simple syntax for this function:

grep EXPR, LIST

Return Value
This function returns the number of times the expression returned true in scalar context and list of elements that matched the expression in list context.

Example
Following is the example code showing its basic usage:

#!/usr/bin/perl

@list = (1,"Test", 0, "foo", 20 );
@has_digit = grep (/\d/, @list);

print "@has_digit\n";

When above code is executed, it produces the following result:

1 0 20

hex

Description
This function interprets EXPR as a hexadecimal string and returns the value, or converts \$_ if EXPR is omitted.

Syntax
Following is the simple syntax for this function:

hex EXPR

hex

Return Value
This function returns numeric value equivalent to hexa in scalar context.

Example
Following is the example code showing its basic usage:

#!/usr/bin/perl

print hex '0xAF'; # prints '175'
print '\n';
print hex 'aF'; # same

When above code is executed, it produces the following result:

175
**import**

**Description**
This function is an ordinary method (subroutine) defined (or inherited) by modules that wish to export names to another module. The use function calls the import method for the package used.

**Syntax**
Following is the simple syntax for this function:

```
import LIST
```

**Return Value**
This function does not return any value.

**Example**
Following is the example code showing its basic usage:

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

package Util;
use base 'Exporter';

our @EXPORT_OK = ('foo', 'bar');

sub foo {
    print "foo!";
}
sub bar {
    print "bar!";
}

package Amy;
use Util 'foo';  # only import foo()
```
foo(); # works fine
bar(); # blows up

When above code is executed, it produces the following result:

index

Description
This function returns the position of the first occurrence of SUBSTR in STR, starting at the beginning (starting at zero), or from POSITION if specified.

Syntax
Following is the simple syntax for this function:

index STR, SUBSTR, POSITION

index STR, SUBSTR

Return Value
This function returns -1 on failure otherwise Position of matching string (starting at zero for the first character).

Example
Following is the example code showing its basic usage:

#!/usr/bin/perl

$string = "That is test";

$index = index ($string, 'is');

print "Position of is in the string $index\n";

When above code is executed, it produces the following result:

Position of is in the string 5
### int

**Description**
This function returns the integer element of EXPR, or \$_ if omitted. The int function does not do rounding. If you need to round a value up to an integer, you should use sprintf.

**Syntax**
Following is the simple syntax for this function:

```
int EXPR
```

**Return Value**
This function returns the integer part of EXPR.

**Example**
Following is the example code showing its basic usage:

```
#!/usr/bin/perl

$int_val = int( 6.23930 );
print"Integer value is $int_val\n";

$int_val = int( -6.23930 );
print"Integer value is $int_val\n";

$int_val = int( 10 / 3 );
print"Integer value is $int_val\n";
```

When above code is executed, it produces the following result:

```
Integer value is 6
Integer value is -6
Integer value is 3
```
ioctl

Description
This function performs the function FUNCTION using the system function ioctl( ), using SCALAR to set or receive information when appropriate. The available values for FUNCTION are completely system independent. You should refer to your ioctl.h C header file, if you have one available, for suitable values.

Syntax
Following is the simple syntax for this function:

    ioctl FILEHANDLE, FUNCTION, SCALAR

Return Value
This function returns undef on failure otherwise 0 but true if the return value from the ioctl( ) is 0.

Example
Following is the example code showing its basic usage:

When above code is executed, it produces the following result:

join

Description
This function combines the elements of LIST into a single string using the value of EXPR to separate each element. It is effectively the opposite of split.

Note that EXPR is only interpolated between pairs of elements in LIST; it will not be placed either before the first or after the last element in the string. To join together strings without a separator, supply an empty string rather than undef.

Syntax
Following is the simple syntax for this function:

    join EXPR, LIST
**Return Value**

This function returns the joined string.

**Example**

Following is the example code showing its basic usage:

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

$string = join( "-", "one", "two", "three" );
print"Joined String is $string\n";

$string = join( "", "one", "two", "three" );
print"Joined String is $string\n";
```

When above code is executed, it produces the following result:

```
Joined String is one-two-three
Joined String is onetwothree
```

**keys**

**Description**

This function returns all the keys of the HASH as a list. The keys are returned in random order but, in fact, share the same order as that used by values and each.

**Syntax**

Following is the simple syntax for this function:

```
keys HASH
```

**Return Value**

This function returns number of keys in the hash in scalar context and list of keys in list context.

**Example**

Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl
```
%hash = ('One' => 1,
        'Two' => 2,
        'Three' => 3,
        'Four' => 4);

@values = values( %hash );
print("Values are ", join("-", @values), "\n");

@keys = keys( %hash );
print("Keys are ", join("-", @keys), "\n");

When above code is executed, it produces the following result:

Values are 4-3-2-1
Keys are Four-Three-Two-One

**kill**

**Description**

This function sends a signal to a list of processes. Returns the number of processes successfully signaled.

If SIGNAL is zero, no signal is sent to the process. This is a useful way to check that a child process is alive and hasn't changed its UID. The precise list of signals supported is entirely dependent on the system implementation:

<table>
<thead>
<tr>
<th>Name</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td>Aborts the process</td>
</tr>
<tr>
<td>SIGARLM</td>
<td>Alarm signal</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>Arithmetic exception</td>
</tr>
<tr>
<td>SIGHUP</td>
<td>Hang up</td>
</tr>
<tr>
<td>SIGILL</td>
<td>Illegal instruction</td>
</tr>
<tr>
<td>SIGINT</td>
<td>Interrupt</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>Termination signal</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>Write to a pipe with no readers.</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>Quit signal.</td>
</tr>
</tbody>
</table>
### SIGSEGV
Segmentation fault

### SIGTERM
Termination signal

### SIGUSER1
Application-defined signal 1

### SIGUSER2
Application-defined signal 2

### Syntax
Following is the simple syntax for this function:

```
kill EXPR, LIST
```

### Return Value
This function returns the number of processes successfully signaled.

### Example
Following is the example code showing its basic usage:

```
#!/usr/bin/perl

$cnt = kill 0, getppid(), getpgrp(), 2000;

print "Signal sent to $cnt process\n";
```

When above code is executed, it produces the following result:

### last

### Description
This is not a function. The last keyword is a loop-control statement that immediately causes the current iteration of a loop to become the last. No further statements are executed, and the loop ends. If LABEL is specified, then it drops out of the loop identified by LABEL instead of the currently enclosing loop.

### Syntax
Following is the simple syntax for this function:

```
last LABEL
```
last

**Return Value**

This does not return any value.

**Example**

Following is the example code showing its basic usage:

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

$count = 0;

while( 1 ){
    $count = $count + 1;
    if( $count > 4 ){
        print "Going to exist out of the loop\n";
        last;
    }else{
        print "Count is $count\n";
    }
}

print "Out of the loop\n";
```

When above code is executed, it produces the following result:

```
Count is 1
Count is 2
Count is 3
Count is 4
Going to exist out of the loop
Out of the loop
```
lc

Description
This function returns a lowercased version of EXPR, or $_ if EXPR is omitted.

Syntax
Following is the simple syntax for this function:

```
lc EXPR
```

Return Value
This function returns a lowercased version of EXPR, or $_ if EXPR is omitted.

Example
Following is the example code showing its basic usage:

```
#!/usr/bin/perl

$orig_string = "This is Test and CAPITAL";
$changed_string = lc( $orig_string );

print "Changed String is : $changed_string\n";
```

When above code is executed, it produces the following result:

```
Changed String is : this is test and capital
```

lcfirst

Description
This function returns the string EXPR or $_ with the first character lowercased.

Syntax
Following is the simple syntax for this function:
lcfirst EXPR

lfirstc

**Return Value**
This function returns the string EXPR or $ _ with the first character lowercased.

**Example**
Following is the example code showing its basic usage:

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

$orig_string = "This is Test and CAPITAL";
$changed_string = lcfirst( $orig_string ) ;

print "Changes String is : $changed_string\n";
```

When above code is executed, it produces the following result:

Changes String is : this is Test and CAPITAL

**length**

**Description**
This function returns the length, in characters, of the value of EXPR, or $ _ if not specified. Use scalar context on an array or hash if you want to determine the corresponding size.

**Syntax**
Following is the simple syntax for this function:

```perl
length EXPR
```

**Return Value**
This function returns the size of string.
Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl
$orig_string = "This is Test and CAPITAL";
$string_len = length( orig_string );
print "Length of String is : $string_len\n";
```

When above code is executed, it produces the following result:

```
Length of String is : 11
```

link

Description
This function creates a new file name, NEWFILE, linked to the file OLDFILE. The function creates a hard link; if you want a symbolic link, use the symlink function.

Syntax
Following is the simple syntax for this function:

```
link OLDFILE,NEWFILE
```

Return Value
This function returns 0 on failure and 1 on success.

Example
Following is the example code showing its basic usage, this will create new file using existing file:

```perl
#!/usr/bin/perl
$existing_file = "/usr/home/test1";
$new_file = "/usr/home/test2";
$retval = link $existing_file, $new_file ;
```
if( $retval == 1 ){
    print"Link created successfully\n";
}else{
    print"Error in creating link $!\n";
}

When above code is executed, it produces the following result:

Link created successfully

listen

Description
This function configures the network socket SOCKET for listening to incoming network connections. Sets the incoming connection queue length to EXPR. You might want to consider using the IO::Socket module, which provides a much easier way of creating and listening to network sockets.

Syntax
Following is the simple syntax for this function:

listen SOCKET, EXPR

Return Value
This function returns 0 on failure and 1 on success.

Example
Following is the example code showing its basic usage, this is a server example alongwith socket implementation Perl Socket:

Perl Socket

#!/usr/bin/perl -w
# server.pl
#------------------

use strict;
use Socket;
# use port 7890 as default
my $port = shift || 7890;
my $proto = getprotobynumber('tcp');

# create a socket, make it reusable
socket(SOCKET, PF_INET, SOCK_STREAM, $proto)
    or die "Can't open socket $!
">
setsockopt(SOCKET, SOL_SOCKET, SO_REUSEADDR, 1)
    or die "Can't set socket option to SO_REUSEADDR $!
">

# bind to a port, then listen
bind( SOCKET, pack( 'Sn4x8', AF_INET, $port, "\0\0\0\0" ))
    or die "Can't bind to port $port! 
">
listen(SOCKET, 5) or die "listen: $!";
print "SERVER started on port $port\n";

# accepting a connection
my $client_addr;
while ($client_addr = accept(NET_SOCKET, SOCKET)) {
    # send them a message, close connection
    print NEW_SOCKET "Smile from the server";
    close NEW_SOCKET;
}

When above code is executed, it produces the following result:

local

Description

This function sets the variables in LIST to be local to the current execution block. If more than one value is specified, you must use parentheses to define the list.
Note that `local` creates a local copy of a variable, which then goes out of scope when the enclosing block terminates. The localized value is then used whenever it is accessed, including any subroutines and formats used during that block.

### Syntax
Following is the simple syntax for this function:

```perl
local LIST
```

### Return Value
This function does not return any value.

### Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

local $foo;       # make $foo dynamically local
local (@wid, %get); # make list of variables local
local $foo = "flurp"; # make $foo dynamic, and init it
local @oof = @bar; # make @oof dynamic, and init it
```

When above code is executed, it produces the following result:

```perl

```

### localtime

#### Description
This function converts the time specified by `EXPR` in a list context, returning a nine-element array with the time analyzed for the current local time zone. The elements of the array are

```
# 0 1 2 3 4 5 6 7 8
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime(time);
```

If `EXPR` is omitted, uses the value returned by time.

$`mday` is the day of the month, and `$mon` is the month itself, in the range 0..11 with 0 indicating January and 11 indicating December.
$year is the number of years since 1900, not just the last two digits of the year. That is, $year is 123 in year 2023. The proper way to get a complete 4-digit year is simply: $year += 1900;

**Syntax**

Following is the simple syntax for this function:

```perl
localtime EXPR
```

**Return Value**

This function returns a string of the form: Thu Sep 21 14:52:52 2000 in scalar context and the individual time component values (seconds, minutes, hours, day of month, month, year, day of week, day of year, daylight savings time) in list context.

**Example**

Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w
use POSIX;

($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =
   localtime(time);
$year += 1900;
print "$sec, $min, $hour, $mday, $mon, $year, $wday, $yday, $isdst\n";
$now_string = localtime;
print "$now_string\n";

$now_string = strftime "%a %b %e %H:%M:%S %Y", localtime;
print "$now_string\n";
```

When above code is executed, it produces the following result:

```plaintext
19, 58, 14, 1, 8, 2013, 0, 243, 0
Sun Sep  1 14:58:19 2013
Sun Sep  1 14:58:19 2013
```
lock

Description
This function places an advisory lock on a shared variable, or referenced object contained in THING until the lock goes out of scope.

lock() is a "weak keyword" : this means that if you've defined a function by this name before any calls to it, that function will be called instead.

Syntax
Following is the simple syntax for this function:

```
lock THING
```

Return Value
This function does not return any value.

Example
Following is the example code showing its basic usage:

```

```
When above code is executed, it produces the following result:

```

```

log

Description
This function returns the natural logarithm of EXPR, or $_ if omitted. To get the log of another base, use basic algebra: The base-N log of a number is equal to the natural log of that number divided by the natural log of N.

Syntax
Following is the simple syntax for this function:

```
log EXPR
```

```
log
```
Return Value
This function returns Floating point number in scalar context.

Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

print "log10(2): ", log10(2), "\n";
print "log10(2): ", log10(3), "\n";
print "log10(2): ", log10(5), "\n";

sub log10 {
    my $n = shift;
    return log($n)/log(10);
}
```

When above code is executed, it produces the following result:

```
log10(2): 0.301029995663981
log10(2): 0.477121254719662
log10(2): 0.698970004336019
```

lstat

Description
This function performs the same tests as the stat function on FILEHANDLE or the file referred to by EXPR or $_.

If the file is a symbolic link, it returns the information for the link, rather than the file it points to. Otherwise, it returns the information for the file.

Syntax
Following is the simple syntax for this function:

```
lstat FILEHANDLE
lstat EXPR
```
Return Value

This function returns a list of 13 elements in list context, these fields are as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>dev</td>
</tr>
<tr>
<td>1</td>
<td>ino</td>
</tr>
<tr>
<td>2</td>
<td>mode</td>
</tr>
<tr>
<td>3</td>
<td>nlink</td>
</tr>
<tr>
<td>4</td>
<td>uid</td>
</tr>
<tr>
<td>5</td>
<td>gid</td>
</tr>
<tr>
<td>6</td>
<td>rdev</td>
</tr>
<tr>
<td>7</td>
<td>size</td>
</tr>
<tr>
<td>8</td>
<td>atime</td>
</tr>
<tr>
<td>9</td>
<td>mtime</td>
</tr>
<tr>
<td>10</td>
<td>ctime</td>
</tr>
<tr>
<td>11</td>
<td>blksize</td>
</tr>
<tr>
<td>12</td>
<td>blocks</td>
</tr>
</tbody>
</table>

**NOTE:** The epoch was at 00:00 January 1, 1970 GMT.

Example

Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

$filename = "/tmp/test.pl";
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,
  $atime,$mtime,$ctime,$blksize,$blocks) = lstat($filename);
printf "File is %s, \n size is %s, \n perm %04o, mtime %s\n",
  $filename, $size, $mode & 07777,
  scalar localtime $mtime;
```

When above code is executed, it produces the following result:
m

Description
This match operator is used to match any keyword in given expression. Parentheses after initial m can be any character and will be used to delimit the regular expression statement.

Regular expression variables include $, which contains whatever the last grouping match matched; $&, which contains the entire matched string; $`, which contains everything before the matched string; and $', which contains everything after the matched string.

Syntax
Following is the simple syntax for this function:

m//

Return Value
This function returns 0 on failure and 1 on success,

Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

$string = "The food is in the salad bar";
$string =~ m/foo/;
print "Before: $`\n";
print "Matched: $&\n";
print "After: $'\n";
```

When above code is executed, it produces the following result:

Before: The
Matched: foo
After: d is in the salad bar
map

Description
This function Evaluates EXPR or BLOCK for each element of LIST. For each iteration, $$_holds the value of the current element, which can also be assigned to allow the value of the element to be updated.

Simply, Perl's map() function runs an expression on each element of an array, and returns a new array with the results.

Syntax
Following is the simple syntax for this function:

```
map EXPR, LIST
map BLOCK LIST
```

Return Value
This function returns the total number of elements so generated in scalar context and list of values in list context.

Example
Following is the example code showing its basic usage:

```
#!/usr/bin/perl -w

@myNames = ('jacob', 'alexander', 'ethan', 'andrew');
@ucNames = map(ucfirst, @myNames);

foreach $key ( @ucNames ){
    print "$key\n";
}
```

When above code is executed, it produces the following result:

```
Jacob
Alexander
Ethan
```
**mkdir**

**Description**
This function makes a directory with the name and path `EXPR` using the mode specified by `MODE`, which should be supplied as an octal value for clarity.

**Syntax**
Following is the simple syntax for this function:

```perl
mkdir EXPR,MODE
```

**Return Value**
This function returns 0 on failure and 1 on success.

**Example**
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

$dirname = "/tmp/testdir";
mkdir $dirname, 0755;
```

When above code is executed, it produces the following result in `/tmp` directory:

```
  drwxr-xr-x  2 root root  4096 Sep 08 11:55 testdir
```

**msgctl**

**Description**
This function calls the system function msgctl( ) with the arguments `ID`, `CMD`, and `ARG`. You may need to include the IPC::SysV package to obtain the correct constants.

**Syntax**
Following is the simple syntax for this function:
msgctl ID, CMD, ARG

**Return Value**

This function returns 0 but true if the system function returns 0 and 1 on success.

**Example**

Following is the example code showing its basic usage:

When above code is executed, it produces the following result:

---

**msgget**

**Description**

This function calls the System V IPC function msgget(2). Returns the message queue id, or the undefined value if there is an error.

**Syntax**

Following is the simple syntax for this function:

```
msgget KEY, FLAGS
```

**Return Value**

This function returns undef on error and Message queue ID on success.

**Example**

Following is the example code showing its basic usage:

When above code is executed, it produces the following result:
**msgrcv**

**Description**
This function receives a message from the queue ID, placing the message into the variable VAR, up to a maximum size of SIZE.

**Syntax**
Following is the simple syntax for this function:

```
msgrcv ID, VAR, SIZE, TYPE, FLAGS
```

**Return Value**
This function returns 0 on error and 1 on success.

**Example**
Following is the example code showing its basic usage:

When above code is executed, it produces the following result:

**msgsnd**

**Description**
This function sends the message MSG to the message queue ID, using the optional FLAGS.

**Syntax**
Following is the simple syntax for this function:

```
msgsnd ID, MSG, FLAGS
```

**Return Value**
This function returns 0 on error and 1 on success.

**Example**
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

my $string = "We are the world";
print "$string\n";
myfunction();
print "$string\n";

sub myfunction
{
  my $string = "We are the function";
  print "$string\n";
  mysub();
}
```

When above code is executed, it produces the following result:

```plaintext
We are the world
We are the function
```

---

### my

#### Description

This function declares the variables in LIST to be lexically scoped within the enclosing block. If more than one variable is specified, all variables must be enclosed in parentheses.

#### Syntax

Following is the simple syntax for this function:

```perl
my LIST
```

#### Return Value

This function does not return any value.

#### Example

Following is the example code showing its basic usage:
When above code is executed, it produces the following result:

```
We are the world
We are the function
We are the world
We are the world
```

### next

#### Description

This is not a function, it causes the current loop iteration to skip to the next value or next evaluation of the control statement. No further statements in the current loop are executed. If LABEL is specified, then execution skips to the next iteration of the loop identified by LABEL.

#### Syntax

Following is the simple syntax for this function:

```
next LABEL
```

#### Return Value

This function does not return any value.

#### Example

Following is the example code showing its basic usage:

```
#!/usr/bin/perl -w

@list = (1,2,3,4,5,5,3,6,7,1);
```
```perl
foreach $key (@list) {
    if ($key == 5) {
        next;
    } else {
        print "Key value is $key\n";
    }
}
```

When above code is executed, it produces the following result:

```
Key value is 1
Key value is 2
Key value is 3
Key value is 4
Key value is 3
Key value is 6
Key value is 7
Key value is 1
```

**no**

**Description**

This function calls the unimport function defined in MODULE to unimport all symbols from the current package if MODULE supports it, or only the symbols referred to by LIST. It can be said that no is opposite of import.

**Syntax**

Following is the simple syntax for this function:

```perl
no Module VERSION LIST

no Module VERSION

no MODULE LIST
```
Return Value
This function does not return any value.

Example
Following is the example code showing its basic usage:

When above code is executed, it produces the following result:

oct

Description
This function converts EXPR from octal to decimal. For example, oct('0760') will return '496'. You can use the string returned as a number because Perl will automatically convert strings to numbers in numeric contexts. Passed parameter should be an octal number otherwise it will produce zero as a result.

Syntax
Following is the simple syntax for this function:

```
oct EXPR
```

Return Value
This function returns the decimal value.

Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

print("oct(88) ", oct('88'), "\n");
```
print("oct(0760) ", oct('0760'), "\n");

When above code is executed, it produces the following result:

genric

**open**

**Description**

This function opens a file using the specified file handle. The file handle may be an expression, the resulting value is used as the handle. If no filename is specified a variable with the same name as the file handle used (this should be a scalar variable with a string value referring to the file name). The special file name '-' refers to STDIN and '->' refers to STDOUT.

**Syntax**

Following is the simple syntax for this function:

```perl
open FILEHANDLE, EXPR, LIST
open FILEHANDLE, EXPR
open FILEHANDLE
```

**Return Value**

This function returns 0 on failure and 1 on success.

**Example**

Following is the syntax to open file.txt in read-only mode. Here less than < sign indicates that file has to be opened in read-only mode

```perl
open(DATA, "<file.txt");
```

Here DATA is the file handle which will be used to read the file. Here is the example which will open a file and will print its content over the screen.

```perl
#!/usr/bin/perl
open(DATA, "<file.txt");
```
```perl
while(<DATA>)
{
    print "$_";
}
```

Following is the syntax to open file.txt in writing mode. Here less than > sign indicates that file has to be opened in writing mode.

```perl
open(DATA, ">file.txt");
```

This example actually truncates (empties) the file before opening it for writing, which may not be the desired effect. If you want to open a file for reading and writing, you can put a plus sign before the > or < characters.

For example, to open a file for updating without truncating it:

```perl
open(DATA, "+<file.txt");
```

To truncate the file first:

```perl
open DATA, ">+file.txt" or die "Couldn't open file file.txt, $!";
```

You can open a file in append mode. In this mode writing point will be set to the end of the file

```perl
open(DATA,">>file.txt") || die "Couldn't open file file.txt, $!";
```

A double >> opens the file for appending, placing the file pointer at the end, so that you can immediately start appending information. However, you can't read from it unless you also place a plus sign in front of it:

```perl
open(DATA,"+>>file.txt") || die "Couldn't open file file.txt, $!";
```

Following is the table which gives the possible values of different modes.

<table>
<thead>
<tr>
<th>Entities</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; or r</td>
<td>Read Only Access</td>
</tr>
<tr>
<td>&gt; or w</td>
<td>Creates, Writes, and Truncates</td>
</tr>
<tr>
<td>&gt;&gt; or a</td>
<td>Writes, Appends, and Creates</td>
</tr>
<tr>
<td>+&lt; or r+</td>
<td>Reads and Writes</td>
</tr>
<tr>
<td>+&gt; or w+</td>
<td>Reads, Writes, Creates, and Truncates</td>
</tr>
<tr>
<td>+&gt;&gt; or a+</td>
<td>Reads, Writes, Appends, and Creates</td>
</tr>
</tbody>
</table>
opendir

**Description**
This function opens the directory EXPR, associating it with DIRHANDLE for processing, using the readdir function.

**Syntax**
Following is the simple syntax for this function:

```perl
opendir DIRHANDLE, EXPR
```

**Return Value**
This function returns true if successful.

**Example**
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

$dirname = "/tmp";

opendir ( DIR, $dirname ) || die "Error in opening dir $dirname\n";
while( ($filename = readdir(DIR)) ){
  print("$filename\n");
}
closedir(DIR);
```

When above code is executed, it produces the following result:

```
.
..
testdir
```
ord

Description
This function returns the ASCII numeric value of the character specified by EXPR, or \$_ if omitted. For example, ord('A') returns a value of 65.

Syntax
Following is the simple syntax for this function:

```
ord EXPR
```

Return Value
This function returns Integer.

Example
Following is the example code showing its basic usage:

```
#!/usr/bin/perl -w

print("ord() ", ord('G'), "\n");
```

When above code is executed, it produces the following result:

```
ord() 71
```

our

Description
This function defines the variables specified in LIST as being global within the enclosing block, file, or eval statement. It is effectively the opposite of my.it declares a variable to be global within the entire scope, rather than creating a new private variable of the same name. All other options are identical to my;

An our declaration declares a global variable that will be visible across its entire lexical scope, even across package boundaries. The package in which the variable is entered is determined at the point of the declaration, not at the point of use. If more than one value is listed, the list must be placed in parentheses.
Syntax

Following is the simple syntax for this function:

```
our EXPR

our EXPR TYPE

our EXPR : ATTRS

our TYPE EXPR : ATTRS
```

Return Value

This function does not return any value.

Example

Following is the example code showing its basic usage:

```
#!/usr/bin/perl -w

our $string = "We are the world";
print "$string\n";
myfunction();
print "$string\n";

sub myfunction
{
  our $string = "We are the function";
  print "$string\n";
}
```

When above code is executed, it produces the following result:

```
We are the world
We are the function
We are the function
```
**pack**

**Description**

This function evaluates the expressions in LIST and packs them into a binary structure specified by EXPR. The format is specified using the characters shown in Table below:

Each character may be optionally followed by a number, which specifies a repeat count for the type of value being packed. That is nibbles, chars, or even bits, according to the format. A value of * repeats for as many values remain in LIST. Values can be unpacked with the unpack function.

For example, a5 indicates that five letters are expected. b32 indicates that 32 bits are expected. h8 indicates that 8 nybbles (or 4 bytes) are expected. P10 indicates that the structure is 10 bytes long.

**Syntax**

Following is the simple syntax for this function:

```
pack EXPR, LIST
```

**Return Value**

- This function returns a packed version of the data in LIST using TEMPLATE to determine how it is coded.

Here is the table which gives values to be used in TEMPLATE.

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>ASCII character string padded with null characters</td>
</tr>
<tr>
<td>A</td>
<td>ASCII character string padded with spaces</td>
</tr>
<tr>
<td>b</td>
<td>String of bits, lowest first</td>
</tr>
<tr>
<td>B</td>
<td>String of bits, highest first</td>
</tr>
<tr>
<td>c</td>
<td>A signed character (range usually -128 to 127)</td>
</tr>
<tr>
<td>C</td>
<td>An unsigned character (usually 8 bits)</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>d</td>
<td>A double-precision floating-point number</td>
</tr>
<tr>
<td>f</td>
<td>A single-precision floating-point number</td>
</tr>
<tr>
<td>h</td>
<td>Hexadecimal string, lowest digit first</td>
</tr>
<tr>
<td>H</td>
<td>Hexadecimal string, highest digit first</td>
</tr>
<tr>
<td>i</td>
<td>A signed integer</td>
</tr>
<tr>
<td>I</td>
<td>An unsigned integer</td>
</tr>
<tr>
<td>l</td>
<td>A signed long integer</td>
</tr>
<tr>
<td>L</td>
<td>An unsigned long integer</td>
</tr>
<tr>
<td>n</td>
<td>A short integer in network order</td>
</tr>
<tr>
<td>N</td>
<td>A long integer in network order</td>
</tr>
<tr>
<td>p</td>
<td>A pointer to a string</td>
</tr>
<tr>
<td>s</td>
<td>A signed short integer</td>
</tr>
<tr>
<td>S</td>
<td>An unsigned short integer</td>
</tr>
<tr>
<td>u</td>
<td>Convert to uuencode format</td>
</tr>
<tr>
<td>v</td>
<td>A short integer in VAX (little-endian) order</td>
</tr>
<tr>
<td>V</td>
<td>A long integer in VAX order</td>
</tr>
<tr>
<td>x</td>
<td>A null byte</td>
</tr>
<tr>
<td>X</td>
<td>Indicates &quot;go back one byte&quot;</td>
</tr>
<tr>
<td>@</td>
<td>Fill with nulls (ASCII 0)</td>
</tr>
</tbody>
</table>
Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

$bits = pack("c", 65);
# prints A, which is ASCII 65.
print "bits are $bits";
$bits = pack("x");
# $bits is now a null character.
print "bits are $bits";
$bits = pack("sai", 255, "T", 30);
# creates a seven character string on most computers'
print "bits are $bits"

@array = unpack("sai", "$bits");

#Array now contains three elements: 255, T and 30.
print "Array $array[0]"
print "Array $array[1]"
print "Array $array[2]"
```

When above code is executed, it produces the following result:

```
bits are A
bits are T
bits are T
Array 255
Array T
Array 30
```
package

Description
This function Changes the name of the current symbol table to NAME. The scope of the package name is until the end of the enclosing block. If NAME is omitted, there is no current package, and all function and variables names must be declared with their fully qualified names.

Syntax
Following is the simple syntax for this function:

```
package NAME
package
```

Return Value
This function does not return any value.

Example
To understand package keyword check Perl Modules session.

Following is the example code showing its basic usage:

When above code is executed, it produces the following result:

pipe

Description
This function opens a pair of connected communications pipes: READHANDLE for reading and WRITEHANDLE for writing. You may need to set $| to flush your WRITEHANDLE after each command.

Syntax
Following is the simple syntax for this function:
pipe READHANDLE, WRITEHANDLE

**Return Value**
This function returns 0 on failure and 1 on success.

**Example**
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

use IO::Handle;

pipe(PARENTREAD, PARENTWRITE);
pipe(CHILDREAD, CHILDWRITE);

PARENTWRITE->autoflush(1);
CHILDWRITE->autoflush(1);

if ($child = fork) # Parent code
{
  close CHILDREAD; # We don't need these in the parent
  close PARENTWRITE;
  print CHILDWRITE "34+56;\n";
  chomp($result = <PARENTREAD>);
  print "Got a value of $result from child\n";
  close PARENTREAD;
  close CHILDWRITE;
  waitpid($child,0);
}else{
  close PARENTREAD; # We don't need these in the child
  close CHILDWRITE;
  chomp($calculation = <CHILDREAD>);
  print "Got $calculation\n";
  $result = eval "$calculation"
  print PARENTWRITE "$result\n";
```
It will produce following results: You can see that the calculation is sent to CHILDWRITE, which is then read by the child from CHILDREAD. The result is then calculated and sent back to the parent via PARENTWRITE, where the parent reads the result from PARENTREAD.

Got 34+56;
Got a value of 90 from child

pop

Description
This function returns the last element of ARRAY, removing the value from the array. Note that ARRAY must explicitly be an array, not a list.

If ARRAY is omitted, it pops the last value from @ARGV in the main program or when called within eval STRING, or the BEGIN, CHECK, INIT, or END blocks. Otherwise, it attempts to pop information from the @_ array within a subroutine. It is the opposite of push, which when used in combination, allows you to implement "stacks".

Note that after applying pop the array will be shortened by one element.

Syntax
Following is the simple syntax for this function:

pop ARRAY

Return Value
This function returns undef if list is empty else last element from the array.

Example
Following is the example code showing its basic usage:
#!/usr/bin/perl -w

@a = (1, 2, 3, 4);
print("pop( ), pop(@a), " leaves ",@a, "\n");

When above code is executed, it produces the following result:

pop( ) 4 leaves 123

pos

Description
This function is used to find the offset or position of the last matched substring. If SCALAR is specified, it will return the offset of the last match on that scalar variable.

You can also assign a value to this function (for example, pos($foo) = 20;) in order to change the starting point of the next match operation.

Offset is counter starting from zeroth position.

Syntax
Following is the simple syntax for this function:

pos EXPR

pos

Return Value
This function returns Integer in Scalar context and then positions of all the matches within the regular expression in List context.

Example
Following is the example code showing its basic usage:

#!/usr/bin/perl -w

$name = "This is alpha beta gamma";
$name =~ m/alpha/g;
print("pos() ", pos($name), ",\n");

When above code is executed, it produces the following result:

pos() 13

print

**Description**

This function prints the values of the expressions in LIST to the current default output filehandle, or to the one specified by FILEHANDLE.

If set, the $\ variable will be added to the end of the LIST.

If LIST is empty, the value in $_ is printed instead.

print accepts a list of values and every element of the list will be interpreted as an expression.

**Syntax**

Following is the simple syntax for this function:

```
print FILEHANDLE LIST

print LIST

print
```

**Return Value**

This function returns 0 on failure and 1 on success.

**Example**

Following is the example code showing its basic usage:

```
#!/usr/bin/perl -w

$string = "That is test";
@list = (1,2,3,4,5,6,);
```
```perl
$index = index ($string, 'is');
print "Position of is in the string $index\n";
print "Print list @list\n";
```

When above code is executed, it produces the following result:

```
Position of is in the string 5
Print list 1 2 3 4 5 6
```

### printf

**Description**

This function prints the value of LIST interpreted via the format specified by FORMAT to the current output filehandle, or to the one specified by FILEHANDLE.

Effectively equivalent to print FILEHANDLE sprintf(FORMAT, LIST)

You can use print in place of printf if you do not require a specific output format. Following is the list of accepted formatting conversions.

<table>
<thead>
<tr>
<th>Format</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>%%</td>
<td>A percent sign</td>
</tr>
<tr>
<td>%c</td>
<td>A character with the given ASCII code</td>
</tr>
<tr>
<td>%s</td>
<td>A string</td>
</tr>
<tr>
<td>%d</td>
<td>A signed integer (decimal)</td>
</tr>
<tr>
<td>%u</td>
<td>An unsigned integer (decimal)</td>
</tr>
<tr>
<td>%o</td>
<td>An unsigned integer (octal)</td>
</tr>
<tr>
<td>%x</td>
<td>An unsigned integer (hexadecimal)</td>
</tr>
<tr>
<td>%X</td>
<td>An unsigned integer (hexadecimal using uppercase characters)</td>
</tr>
</tbody>
</table>
%e  A floating point number (scientific notation)

%E  A floating point number, uses E instead of e

%f  A floating point number (fixed decimal notation)

%g  A floating point number (%e or %f notation according to value size)

%G  A floating point number (as %g, but using .E. in place of .e. when appropriate)

%p  A pointer (prints the memory address of the value in hexadecimal)

%n  Stores the number of characters output so far into the next variable in the parameter list

Perl also supports flags that optionally adjust the output format. These are specified between the % and conversion letter. They are shown in the following table:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>space</td>
<td>Prefix positive number with a space</td>
</tr>
<tr>
<td>+</td>
<td>Prefix positive number with a plus sign</td>
</tr>
<tr>
<td>-</td>
<td>Left-justify within field</td>
</tr>
<tr>
<td>0</td>
<td>Use zeros, not spaces, to right-justify</td>
</tr>
<tr>
<td>#</td>
<td>Prefix non-zero octal with .0. and hexadecimal with .0x.</td>
</tr>
<tr>
<td>number</td>
<td>Minimum field width</td>
</tr>
<tr>
<td>.number</td>
<td>Specify precision (number of digits after decimal point) for floating point numbers</td>
</tr>
</tbody>
</table>
I Interpret integer as C-type .long. or .unsigned long.

h Interpret integer as C-type .short. or .unsigned short.

V Interpret integer as Perl's standard integer type

v Interpret the string as a series of integers and output as numbers separated by periods or by an arbitrary string extracted from the argument when the flag is preceded by *.

Syntax
Following is the simple syntax for this function:

```plaintext
printf FILEHANDLE FORMAT, LIST
printf FORMAT, LIST
```

Return Value
This function

Example
Following is the example code showing its basic usage:

```plaintext
#!/usr/bin/perl -w
printf "%d\n", 3.1415126;
printf "The cost is \$%6.2f\n", 499;
printf "Perl's version is v%vd\n", ^V;
printf "%04d\n", 20;
```

When above code is executed, it produces the following result:

3
The cost is $499.00
Perl's version is v
0020
**prototype**

**Description**
This function returns a string containing the prototype of the function or reference specified by EXPR, or undef if the function has no prototype.

You can also use this to check the availability of built-in functions.

**Syntax**
Following is the simple syntax for this function:

```
prototype EXPR
```

**Return Value**
This function returns undef if no function prototype else returns string containing the prototype of the function or reference specified by EXPR.

**Example**
Following is the example code showing its basic usage:

```
#!/usr/bin/perl -w

$func_prototype = prototype ( "myprint" );
print "myprint prototype is $func_prototype\n";

sub myprint($$){
    print "This is test\n";
}
```

When above code is executed, it produces the following result:

```
myprint prototype is $$
```

**push**

**Description**
This function pushes the values in LIST onto the end of the list ARRAY. Used with pop to implement stacks.
Syntax
Following is the simple syntax for this function:

```
push ARRAY, LIST
```

Return Value
This function returns number of elements in new array.

Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

$, = "",;
$array = ( 1, 2 );
print "Before pushing elements @array \n";
push(@array, (3, 4, 5));
print "After pushing elements @array \n";
```

When above code is executed, it produces the following result:

```
Before pushing elements 1 2
After pushing elements 1 2 3 4 5
```

q

Description
This function can be used instead of single quotes. This is not really a function, more like an operator, but you'll probably look here if you see it in another programmer's program without remembering what it is. You can actually use any set of delimiters, not just the parentheses.

Syntax
Following is the simple syntax for this function:

```
q( string )
```
Return Value
This function returns a single-quoted string.

Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w
$var = 10;
print(q(This is a single quoted string without interpolation, $var));
```

When above code is executed, it produces the following result:

```
```

Description
This function can be used instead of double quotes. This is not really a function, more like an operator, but you'll probably look here if you see it in another programmer's program without remembering what it is. You can actually use any set of delimiters, not just the parentheses.

Syntax
Following is the simple syntax for this function:

```perl
qq ( string )
```

Return Value
This function returns a double-quoted string.

Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w
$var = 10;
print(qq(This is a single quoted string with interpolation, $var));
```
When above code is executed, it produces the following result:

```
This is a single quoted string with interpolation, 10
```

**qr**

**Description**
This function quotes its STRING as a regular expression. STRING is interpolated the same way as PATTERN in m/PATTERN/

**Syntax**
Following is the simple syntax for this function:

```
qr EXPR
```

**Return Value**
This function returns a Perl value which may be used instead of the corresponding /STRING/ expression.

**Example**
Following is the example code showing its basic usage:

```
$rex = qr/my.STRING/is;
s/$rex/foo/;

is is equivalent to

s/my.STRING/foo/is;
```

When above code is executed, it produces the following result:

```
$re = qr/$pattern/;
$string =~ /foo${re}bar/;  # can be interpolated in other patterns
$string =~ $re;  # or used standalone
$string =~ /$re/;  # or this way
```
**quotemeta**

**Description**
This function escapes all meta-characters in EXPR. For example, quotemeta("AB*..C") returns "AB\*\.C".

**Syntax**
Following is the simple syntax for this function:

```
quotemeta EXPR
```

**Return Value**
This function returns a string with all meta-characters escaped.

**Example**
Following is the example code showing its basic usage:

```
#!/usr/bin/perl -w
print quotemeta("AB\n[.]*");
```

When above code is executed, it produces the following result:

```
AB\n[.]*
```

**qw**

**Description**
This function is a quick way to specify a lot of little single-quoted words. For example, qw(foo bar baz) is equivalent to ('foo', 'bar', 'baz'). Some programmers feel that using qw make Perl scripts easier to read. You can actually use any set of delimiters, not just the parentheses.

Simply you can use qw() to prepare an array as shown in the example below.

**Syntax**
Following is the simple syntax for this function:
\texttt{qw EXPR}

**Return Value**
This function return a list consisting of the element of \texttt{LIST} evaluated as if they were single-quoted.

**Example**
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

@array = qw(This is a list of words without interpolation);

foreach $key (@array){
    print "Key is $key\n";
}
```

When above code is executed, it produces the following result:

```
Key is This
Key is is
Key is a
Key is list
Key is of
Key is words
Key is without
Key is interpolation
```

\texttt{qx}

**Description**
This function is an alternative to using back-quotes to execute system commands. For example, \texttt{qx(ls -l)} will execute the UNIX \texttt{ls} command using the -l command-line option. You can actually use any set of delimiters, not just the parentheses.

**Syntax**
Following is the simple syntax for this function:
qx EXPR

**Return Value**

This function returns the value from the executed system command.

**Example**

Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

# summarize disk usage for the /tmp directory
# and store the output of the command into the
# @output array.
@output = qx(du -s /tmp);

print "@output\n";
```

When above code is executed, it produces the following result:

```
176 /tmp
```

**rand**

**Description**

This function returns a random fractional number between 0 and the positive number `EXPR`, or 1 if not specified. Automatically calls `srand` to seed the random number generator unless it has already been called.

**Syntax**

Following is the simple syntax for this function:

```perl
rand EXPR
```

```
rand
```

**Return Value**

This function returns a Floating point number.
Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

# print a random number between 0 and 10.
print("Random Number:", rand(10), "\n");
```

When above code is executed, it produces the following result:

Random Number:2.2592476087539

read

description
This function reads, or attempts to read, LENGTH number of bytes from the file associated with FILEHANDLE into BUFFER. If an offset is specified, the bytes that are read are placed into the buffer starting at the specified offset.

Syntax
Following is the simple syntax for this function:

```perl
read FILEHANDLE, SCALAR, LENGTH, OFFSET
read FILEHANDLE, SCALAR, LENGTH
```

return value
This function the number of bytes read or the undefined value.

Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

my($buffer) = "";
open(FILE, "/etc/services") or
    die("Error reading file, stopped");
while(read(FILE, $buffer, 100))
```
When above code is executed, it produces the following result:

<table>
<thead>
<tr>
<th>Service</th>
<th>Port</th>
<th>Protocol</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>kerberos_master</td>
<td>751</td>
<td>udp</td>
<td>Kerberos authentication</td>
</tr>
<tr>
<td>kerberos_master</td>
<td>751</td>
<td>tcp</td>
<td>Kerberos authentication</td>
</tr>
<tr>
<td>passwd_server</td>
<td>752</td>
<td>udp</td>
<td>Kerberos passwd server</td>
</tr>
</tbody>
</table>

**readdir**

**Description**

This function returns the next directory entry from the directory associated with DIRHANDLE in a scalar context. In a list context, returns all of the remaining directory entries in DIRHANDLE.

**Syntax**

Following is the simple syntax for this function:

```
readdir DIRHANDLE
```

**Return Value**

This function returns the next directory entry from the directory associated with DIRHANDLE in a scalar context. In a list context, returns all of the remaining directory entries in DIRHANDLE.

**Example**

Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

$dirname = "/tmp";

opendir ( DIR, $dirname ) || die "Error in opening dir $dirname\n";
while( ($filename = readdir(DIR))){
```
When above code is executed, it produces the following result:

```
.
..
testdir
```

### readline

#### Description

This function reads a line from the filehandle referred to by EXPR, returning the result. If you want to use a FILEHANDLE directly, it must be passed as a typeglob.

Simply `readline` function is equivalent to `< >`.

#### Syntax

Following is the simple syntax for this function:

```
readline EXPR
```

#### Return Value

This function returns only one line in a scalar context and in a list context, a list of line up to end-of-file is returned.

#### Example

Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

my($buffer) = "";
open(FILE, "/etc/services") or 
    die("Error reading file, stopped");

$buffer = <FILE>;
```
When above code is executed, it produces the following result:

```
# /etc/services:
# $Id: services,v 1.33 2003/03/14 16:41:47 notting Exp $
```

### readlink

#### Description
This function returns the pathname of the file pointed to by the link EXPR, or \$_ if EXPR is not specified.

#### Syntax
Following is the simple syntax for this function:

```
readlink EXPR
```

#### Return Value
This function returns undef on error otherwise pathname of the file.

#### Example
Following is the example code showing its basic usage:

```
#!/usr/bin/perl -w

# assume /tmp/test is a symbolic link on /tmp/usr/test.pl

readlink "/tmp/test";
```
When above code is executed, it produces the following result:

```plaintext
/tmp/usr/test.pl
```

---

### readpipe

#### Description

This function executes EXPR as a command. The output is then returned as a multiline string in scalar text, or with the line returned as individual elements in a list context.

#### Syntax

Following is the simple syntax for this function:

```perl
readpipe EXPR
```

#### Return Value

This function returns String in Scalar Context and returns List in List Context.

#### Example

Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

@result = readpipe( "ls -l /tmp" );

print "@result";
```

When above code is executed, it produces the following result. This will give whole content in /tmp directory:

```plaintext
drwxr-xr-x  2 root   root  4096 Mar 19 11:55 testdir
```

---

### recv

#### Description

This function receives a message on SOCKET attempting to read LENGTH bytes, placing the data read into variable SCALAR. The FLAGS argument takes the same
values as the recvfrom() system function, on which the function is based. When communicating with sockets, this provides a more reliable method of reading fixed-length data than the sysread function or the line-based operator <FH>.

**Syntax**

Following is the simple syntax for this function:

```
recv SOCKET, SCALAR, LEN, FLAGS
```

**Return Value**

This function returns in Scalar Context: undef on error or Number of bytes read.

**Example**

Following is the example code showing its basic usage:

```
When above code is executed, it produces the following result:
```

---

**redo**

**Description**

This function restarts the current loop without forcing the control statement to be evaluated. No further statements in the block are executed. A continue block, if present, will not be executed. If LABEL is specified, execution restarts at the start of the loop identified by LABEL.

**Syntax**

Following is the simple syntax for this function:

```
redo LABEL
```

```
redo
```

**Return Value**

This function does not return any value.
Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w
$c = 1;
$d = 4;
LABEL:
{
    $c++;
    $e = 5;
    redo LABEL if ($c < 3);
    $f = 6;
    last LABEL if ($e > 3);
    $g = 7;
}
$h = 8;
print "$c $d $e $f $g $h\n";
```

When above code is executed, it produces the following result:

```
3 4 5 6 7
```

ref

Description
This function returns a true value if EXPR, or $_ if EXPR is not supplied, is a reference. The actual value returned also defines the type of entity the reference refers to.

The built-in types are:

- REF
- SCALAR
- ARRAY
- HASH
- CODE
Perl

- GLOB
- LVALUE
- IO::Handle

If a variable was blessed with the bless() function, then the new data type will be returned. The new data type will normally be a class name.

**Syntax**

Following is the simple syntax for this function:

```perl
ref EXPR
```

**Return Value**

This function returns empty string if not a reference and string if a reference in Scalar Context.

**Example**

Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w
$foobar = { };
bless($foobar, 'ATMPCLASS');
print "ref() \$foobar is now in class ", ref($foobar), "\n";
```

When above code is executed, it produces the following result:

```
ref() $foobar is now in class ATMPCLASS
```

**rename**

**Description**

This function renames the file with OLDNAME to NEWNAME. Uses the system function rename( ), and so it will not rename files across file systems or volumes. If you want to copy or move a file, use the copy or move command supplied in the File::Copy module.
Syntax
Following is the simple syntax for this function:

```perl
rename OLDNAME, NEWNAME
```

Return Value
This function returns 0 on failure and 1 on success.

Example
First create test file in /tmp directory and then use following code to change file name.

```perl
#!/usr/bin/perl -w

rename("/tmp/test", "/tmp/test2") || die ("Error in renaming");
```

When above code is executed, it produces the following result:

```
the file gets renamed
```

require

Description
This function then it demands that the script requires the specified version of Perl in order to continue if EXPR is numeric. If EXPR or $_ are not numeric, it assumes that the name is the name of a library file to be included. You cannot include the same file with this function twice. The included file must return a true value as the last statement.

This differs from use in that included files effectively become additional text for the current script. Functions, variables, and other objects are not imported into the current name space, so if the specified file includes a package definition, then objects will require fully qualified names.

The specified module is searched for in the directories defined in @INC, looking for a file with the specified name and an extension of .pm.

Syntax
Following is the simple syntax for this function:

```perl
require EXPR
```
require

**Return Value**
This function does not return any value.

**Example**
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

# require to demand a particular perl version.
require 5.003;

# require to include amodule.
require Module;

.............
```

When above code is executed, it produces the following result:

**reset**

**Description**
This function resets (clears) all package variables starting with the letter range specified by EXPR. Generally only used within a continue block or at the end of a loop. If omitted, resets ?PATTERN? matches.

Variables that have been declared using the my() function will not be reset.

Using reset() can reset system variables you may not want to alter-like the ARGV and ENV variables.

**Syntax**
Following is the simple syntax for this function:

```
reset EXPR
```
reset

**Return Value**
This function returns 1.

**Example**
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

my $var = 10;
$van = 5;

print "Var value =$var, Van value =$van\n";
# Now reset all variables who name starts with 'v'
reset('v');
print "Var value =$var, Van value =$van\n";
```

When above code is executed, it produces the following result:

```
Var value =10, Van value =5
Var value =10, Van value =
```

**return**

**Description**
This function returns EXPR at the end of a subroutine, block, or do function. EXPR may be a scalar, array, or hash value; context will be selected at execution time. If no EXPR is given, returns an empty list in list context, undef in scalar context, or nothing in a void context.

**Syntax**
Following is the simple syntax for this function:

```
return EXPR
```
Return Value
This function returns in Scalar Context: List, which may be interpreted as scalar, list, or void context.

Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

$retval = Sum(5,10);
print ("Return value is $retval\n ");

@retval = Sum(5,10);
print ("Return value is @retval\n ");

sub Sum($$){
   my($a, $b) = @_;

   my $c = $a + $b;

   return($a, $b, $c);
}
```

When above code is executed, it produces the following result:

```
Return value is 15
Return value is 5 10 15
```

reverse

Description
This function returns the elements of LIST in reverse order in a list context. In a scalar context, returns a concatenated string of the values of LIST, with all bytes in opposite order.
Syntax
Following is the simple syntax for this function:

```
reverse LIST
```

Return Value
This function returns String in Scalar Context and List in List Context.

Example
Following is the example code showing its basic usage:

```
#!/usr/bin/perl -w

@array = (2,3,4,5,6,7);
print "Reversed Value is ", reverse(@array), "\n";
$string = "Hello World";
print "Reversed Value is ", scalar reverse("$string"), "\n";
```

When above code is executed, it produces the following result:

```
Reversed Value is 765432
Reversed Value is dlroW olleH
```

rewinddir

Description
This function Resets the current position within the directory specified by DIRHANDLE to the beginning of the directory.

Syntax
Following is the simple syntax for this function:

```
rewinddir DIRHANDLE
```

Return Value
This function returns 0 on failure and 1 on success.

Example
Perl

Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

# Open the current directory
opendir(DIR, ".");

# Print all of the directory entries.
print("1st Time: \n");
map( print("$_ \n"), readdir(DIR));
print("\n");

# Print message verifying that there are
# no more directory entries to read.
print("The last file has already been read!\n\n")
  unless readdir(DIR);

# Go back to the beginning.
rewinddir(DIR);

# Print all of the directory entries again.
print("2nd Time: \n");
map( print("$_ \n"), readdir(DIR));
print("\n");
closedir(DIR);
```

When above code is executed, it produces the following result (in /tmp directory):

```
1st Time:
.
..
testdir
The last file has already been read!
2nd Time:
```
rindex

Description
This function operates similar to index, except it returns the position of the last occurrence of SUBSTR in STR. If POSITION is specified, returns the last occurrence at or before that position.

Syntax
Following is the simple syntax for this function:

```perl
rindex STR, SUBSTR, POSITION
rindex STR, SUBSTR
```

Return Value
This function returns undef on failure otherwise position of last occurrence.

Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

$pos = rindex("abcdefghijklmdef", "def");
print "Found position of def $pos\n";

# Use the first position found as the offset to the # next search.
# Note that the length of the target string is # subtracted from the offset to save time.

$pos = rindex("abcdefghijklmdef", "def", $pos-3);
```

When above code is executed, it produces the following result:

```
Found position of def 14
Found position of def 3
```

### rmdir

#### Description
This function deletes the directory specified by EXPR, or `$_` if omitted. Only deletes the directory if the directory is empty.

#### Syntax
Following is the simple syntax for this function:

```
rmdir EXPR
```

#### Return Value
This function returns 0 on failure and 1 on success.

#### Example
Following is the example code showing its basic usage, create one directory testdir inside /tmp:

```
#!/usr/bin/perl -w

rmdir ('/tmp/testdir') || die ('error in deleting directory: $?');
```

When above code is executed, it produces the following result:

```
If directory is empty then it will be deleted
otherwise error message will be generated.
```
S

**Description**
This is not a function. This is the regular expression-substitution operator. Based on the regular expression specified in PATTERN, data is replaced by REPLACE. Like m//, the delimiters are defined by the first character following s.

**Syntax**
Following is the simple syntax for this function:

```
s/PATTERN/REPLACE/
```

**Return Value**
This function returns 0 on failure and number of substitutions made on success.

**Example**
Following is the example code showing its basic usage:

```
#!/usr/bin/perl -w

$string = "This is Test";
# this will replace Test with Best.
$string =~ s/Test/Best/;

print "$string\n";
```

When above code is executed, it produces the following result:

```
This is Best
```

**scalar**

**Description**
This function forces the evaluation of EXPR to be in scalar context, even if it would normally work in list context.
**Syntax**
Following is the simple syntax for this function:

```
scalar EXPR
```

**Return Value**
This function returns Scalar.

**Example**
Following is the example code showing its basic usage:

```
#!/usr/bin/perl -w

@a = (1,2,3,4);
@b = (10,20,30,40);

@c = ( @a, @b );
print "1 - Final Array is@c\n";

@c = ( scalar(@a), scalar(@b) );
print "2 - Final Array is@c\n";
```

When above code is executed, it produces the following result:

```
1 - Final Array is 1 2 3 4 10 20 30 40
2 - Final Array is 4 4
```

**seek**

**Description**
This function moves to a specified position in a file. You can move relative to the beginning of the file (WHENCE = 0), the current position (WHENCE = 1), or the end of the file (WHENCE = 2). This function is mainly used with fixed length records to randomly access specific records of the file.

For WHENCE you may use the constants SEEK_SET , SEEK_CUR , and SEEK_END (start of the file, current position, end of the file) from the Fcntl module.
This function is similar to Unix seek() system call.

**Syntax**
Following is the simple syntax for this function:

```
seek FILEHANDLE, POSITION, WHENCE
```

**Return Value**
This function

**Example**
Following is the example code showing its basic usage:

When above code is executed, it produces the following result:

---

**seekdir**

**Description**
This function sets the current position within DIRHANDLE to POS. The value of POS must be a value previously returned by telldir.

seekdir() function is similar to Unix seekdir() system call.

**Syntax**
Following is the simple syntax for this function:

```
seekdir DIRHANDLE, POS
```

**Return Value**
This function returns 0 on failure and 1 on success.

**Example**
Following is the example code showing its basic usage, create one directory testdir inside /tmp:

```
#!/usr/bin/perl -w
```
```perl
opendir(DIR, "/tmp");

print("Position without read : ", telldir(DIR), "\n");

$dir = readdir(DIR);
print("Position after one read : ", telldir(DIR), "\n");
print "$dir\n"
seekdir(DIR, 0);

$dir = readdir(DIR);
print "$dir\n"
print("Position after second read : ", telldir(DIR), "\n");

closedir(DIR);
```

When above code is executed, it produces the following result:

```
Position without read : 0
Position after one read : 1220443271
  test.txt
  test.txt
Position after second read : 1220443271
```

**select**

**Description**

This function sets the default filehandle for output to FILEHANDLE, setting the filehandle used by functions such as print and write if no filehandle is specified. If FILEHANDLE is not specified, then it returns the name of the current default filehandle.

`select (RBITS, WBITS, EBITS, TIMEOUT )` calls the system function `select( )` using the bits specified. The select function sets the controls for handling non-blocking I/O requests. Returns the number of filehandles awaiting I/O in scalar context, or the number of waiting filehandles and the time remaining in a list context.
**Syntax**

Following is the simple syntax for this function:

```perl
select FILEHANDLE

select

select RBITS, WBITS, EBITS, TIMEOUT
```

**Return Value**

This function returns the previous default filehandle if FILEHANDLE specified and Current default filehandle if FILEHANDLE is not specified.

**Example**

Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

open(FILE,">/tmp/t.out");
$oldHandle = select(FILE);
print("This is sent to /tmp/t.out\n");
select($oldHandle);
print("This is sent to STDOUT\n");
```

When above code is executed, it produces the following result:

This is sent to STDOUT

---

**Description**

This function controls a System V semaphore. You will need to import the IPC:SysV module to get the correct definitions for CMD. The function calls the system semctl( ) function.

**Syntax**

Following is the simple syntax for this function:
Return Value
This function returns undef on failure and 0 but true on success.

Example
Following is the example code showing its basic usage, creating a semaphore and incrementing its value:

```perl
#!/usr/bin/perl -w

# Assume this file name is left.pl

use IPC::SysV;

#use these next two lines if the previous use fails.
eval 'sub IPC_CREAT {0001000}' unless defined &IPC_CREAT;
eval 'sub IPC_EXCL {0002000}' unless defined &IPC_EXCL;
eval 'sub IPC_RMID {0}' unless defined &IPC_RMID;

$key = 1066;

$| = 1;
$num = 0;
$flag = 0;

# Create the semaphore
$id = semget ( $key, 1, &IPC_EXCL|&IPC_CREAT|0777 ) or
die "Can't semget: $!";

foreach ( 1..5 ) {
    $op = 0;
    $operation = pack( "s*", $num, $op, $flag );
    semop( $id, $operation ) or die "Can't semop: $! ";
    print "Left....\n";
    sleep 1;
```
Run the above program in background using $left.pl& and write following another program. Here Left sets the semaphore to 2 and Right prints Right and resets the semaphore to 0. This continues until Left finishes its loop after which it destroys the semaphore with semctl(

```perl
#!/usr/bin/perl -w

# Assume this file name is right.pl

$key = 1066;

$| = 1;
$num = 0;
$flags = 0;

# Identify the semaphore created by left.
$id = semget( $key, 1, 0 ) or die ("Can't semgt : $!");

foreach( 1..5){
    $op = -1;
    $operation = pack( "s*", $num, $op, $flags );
    # Add -1 to the semaphore (now 1)
    semop( $id, $operation ) or die "Can't semop $!";
    print "Right....\n";
    sleep 1;
    $operation = pack( "s*", $num, $op, $flags );
    # Add -1 to the semaphore (now 0)
    semop( $id, $operation ) or die "Can't semop $! ";
```
When above code is executed, it produces the following result:

Right....
Left....
Right....
Left....
Right....
Left....
Right....
Left....

semget

Description
This function returns the semaphore ID associated with KEY, using the system function semget() ie. Finds the semaphore associated with KEY.

Syntax
Following is the simple syntax for this function:

```
semget KEY, NSEMS, FLAGS
```

Return Value
This function returns undef on failure and 0 but true on success.

Example
Following is the example code showing its basic usage, creating a semaphore and incrementing its value:

```
#!/usr/bin/perl -w

# Assume this file name is left.pl
```
Perl

use IPC::SysV;

#use these next two lines if the previous use fails.
eval 'sub IPC_CREAT {0001000}' unless defined &IPC_CREAT;
eval 'sub IPC_EXCL {0002000}' unless defined &IPC_EXCL;
eval 'sub IPC_RMID {0}' unless defined &IPC_RMID;

$key = 1066;

$| = 1;
$num = 0;
$flag = 0;

# Create the semaphor
$id = semget ( $key, 1, &IPC_EXCL|&IPC_CREAT|0777 ) or 
die "Can't semget: $!";

foreach ( 1..5 ) {
    $op = 0;
    $operation = pack( "s*", $num, $op, $flag );
    semop( $id, $operation ) or die "Can't semop: $! ";
    print "Left....\n";
    sleep 1;
    $op = 2;
    $operation = pack( "s*", $num, $op, $flag );
    # add 2 to the semaphore ( now 2 )
    semop( $id, $operation ) or die "Can't semop $! ";
}

semctl ( $id, 0, &IPC_RMID, 0 );

Run the above program in background using $left.pl & and write following another program. Here Left sets the semaphore to 2 and Right prints Right and resets the semaphore to 0. This continues until Left finishes its loop after which it destroys the semaphore with semctl()

#!/usr/bin/perl -w
# Assume this file name is right.pl

$key = 1066;

$| = 1;
$num = 0;
$flags = 0;

# Identify the semaphore created by left.
$id = semget( $key, 1, 0 ) or die ("Can't semgt : "$!");

foreach( 1..5){
    $op = -1;
    $operation = pack( "s*", $num, $op, $flags );
    # Add -1 to the semaphore (now 1)
    semop( $id, $operation ) or die " Can't semop $!";
    print "Right....\n";
    sleep 1;
    $operation = pack( "s*", $num, $op, $flags );
    # Add -1 to the semaphore (now 0)
    semop( $id, $operation ) or die "Can't semop $! ";
}

When above code is executed, it produces the following result. Now run right.pl and It will produce following results:

Right....
Left....
Right....
Left....
Right....
Left....
semop

Description
This function performs the semaphore operations defined by OPSTRING on the semaphore ID associated with KEY. OPSTRING should be a packed array of semop structures, and each structure can be generated with.

Syntax
Following is the simple syntax for this function:

```
semop KEY, OPSTRING
```

Return Value
This function returns 0 on failure and 1 on success.

Example
Following is the example code showing its basic usage, creating a semaphore and incrementing its value:

```
#!/usr/bin/perl -w

# Assume this file name is left.pl

use IPC::SysV;

# use these next two lines if the previous use fails.
eval 'sub IPC_CREAT {0001000}' unless defined &IPC_CREAT;
eval 'sub IPC_EXCL {0002000}' unless defined &IPC_EXCL;
eval 'sub IPC_RMID {0}' unless defined &IPC_RMID;

$key = 1066;

$| = 1;
```
$num = 0;
$flag = 0;

# Create the semaphore
$id = semget( $key, 1, &IPC_EXCL|&IPC_CREAT|0777 ) or
die "Can't semget: $!";
foreach ( 1..5 ) { 
  $op = 0;
  $operation = pack( "s*", $num, $op, $flags );
  semop( $id, $operation ) or die "Can't semop: $! ";
  print "Left....\n";
  sleep 1;
  $op = 2;
  $operation = pack( "s*", $num, $op, $flags );
  # add 2 to the semaphore ( now 2 )
  semop( $id, $operation ) or die "Can't semop $! ";
}
semctl( $id, 0, &IPC_RMID, 0 );

Run the above program in background using $left.pl& and write following another program. Here Left sets the semaphore to 2 and Right prints Right and resets the semaphore to 0. This continues until Left finishes its loop after which it destroys the semaphore with semctl()

#!/usr/bin/perl -w

# Assume this file name is right.pl

$key = 1066;

$| = 1;
$num = 0;
$flags = 0;

# Identify the semaphore created by left.
$id = semget( $key, 1, 0 ) or die("Can't semgt : $!");

foreach( 1..5){
    $op = -1;
    $operation = pack( "s*", $num, $op, $flags );
    # Add -1 to the semaphore (now 1)
    semop( $id, $operation ) or die " Can't semop $!";
    print "Right....\n";
    sleep 1;
    $operation = pack( "s*", $num, $op, $flags );
    # Add -1 to the semaphore (now 0)
    semop( $id, $operation ) or die "Can't semop $! ";
}

Now run right.pl and it will produce the following results:

    Right....
    Left....
    Right....
    Left....
    Right....
    Left....
    Right....
    Left....
    Right....
    Left....

send

Description

This function sends a message on SOCKET (the opposite of recv). If the socket is unconnected, you must supply a destination to communicate to with the TO parameter. In this case, the sendto system function is used in place of the system send function.
The FLAGS parameter is formed from the bitwise or of 0 and one or more of the MSG_OOB and MSG_DONTROUTE options. MSG_OOB allows you to send out-of-band data on sockets that support this notion.

The underlying protocol must also support out-of-band data. Only SOCK_STREAM sockets created in the AF_INET address family support out-of-band data. The MSG_DONTROUTE option is turned on for the duration of the operation. Only diagnostic or routing programs use it.

**Syntax**

Following is the simple syntax for this function:

```plaintext
send SOCKET, MSG, FLAGS, TO

send SOCKET, MSG, FLAGS
```

**Return Value**

This function returns undef on failure else Integer, number of bytes sent.

**Example**

Following is the example code showing its basic usage:

```plaintext

```

When above code is executed, it produces the following result:

```plaintext

```

**setgrent**

**Description**

This function Sets (or resets) the enumeration to the beginning of the set of group entries. This function should be called before the first call to getgrent.

**Syntax**

Following is the simple syntax for this function:

```plaintext
setgrent
```
Return Value
This function does not return any value.

Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

while( ($name,$passwd,$gid,$members) = getgrent() ){
    print "Name  = $name\n";
    print "Password  = $passwd\n";
    print "GID  = $gid\n";
    print "Members  = $members\n";
}

setgrent();   # Set the beginnging of the group database;

print "Again reading /etc/passwd file\n";
while( ($name,$passwd,$gid,$members) = getgrent() ){
    print "Name  = $name\n";
    print "Password  = $passwd\n";
    print "GID  = $gid\n";
    print "Members  = $members\n";
}

endpwent;   #claose the database;
```

When above code is executed, it produces the following result:

```
Name  = root
Password  = x
GID  = 0
Members  = root
Name  = bin
```
Password = x
GID = 1
Members = root bin daemon
Name = daemon
Password = x
GID = 2
Members = root bin daemon
Name = sys
Password = x
GID = 3
Members = root bin adm
Name = adm
Password = x
GID = 4
Members = root adm daemon
.
.
.
Name = kvm
Password = x
GID = 36
Members = qemu
Name = qemu
Password = x
GID = 107
Members =
Name = com
Password = x
GID = 501
Members =
Name = webgrp
Password = x
GID = 502
Members = com
Name = railo
Password = x
GID = 495

**sethostent**

**Description**
This function should be called before the first call to gethostent. The STAYOPEN argument is optional and unused on most systems.

As gethostent() retrieves the information for the next line in the host database, then sethostent sets (or resets) the enumeration to the beginning of the set of host entries.

**Syntax**
Following is the simple syntax for this function:

```
sethostent STAYOPEN
```

**Return Value**
This function does not return any value.

**Example**
Following is the example code showing its basic usage:

```
#!/usr/bin/perl

while( ($name, $aliases, $addrtype, $length, @addrs) = gethostent() ){
    print "Name  = $name\n";
    print "Aliases  = $aliases\n";
    print "Addr Type  = $addrtype\n";
    print "Length  = $length\n";
    print "Addrs  = @addrs\n";
}
```
sethostent(1);

while( ($name, $aliases, $addrtype, $length, @addrs) = gethostent() ){
    print "Name  = \$name\n";
    print "Aliases  = \$aliases\n";
    print "Addr Type  = \$addrtype\n";
    print "Length  = \$length\n";
    print "Addrs  = @addrs\n";
}

endhostent(); # Closes the database;

When above code is executed, it produces the following result:

Name  = ip-50-62-147-141.ip.secureserver.net
Aliases  = ip-50-62-147-141 localhost.secureserver.net
localhost.localdomain localhost
Addr Type  = 2
Length  = 4
Addrs  = 
Name  = ip-50-62-147-141.ip.secureserver.net
Aliases  = ip-50-62-147-141 localhost.secureserver.net
localhost.localdomain localhost
Addr Type  = 2
Length  = 4
Addrs  = 

setnetent

Description
This function should be called before the first call to getnetent. The STAYOPEN argument is optional and unused on most systems. As getnetent() retrieves the information from the next line in the network database, then setnetent sets (or resets) the enumeration to the beginning of the set of host entries.
**Syntax**

Following is the simple syntax for this function:

```
setnetent STAYOPEN
```

**Return Value**

This function does not return any value.

**Example**

Following is the example code showing its basic usage:

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

use Socket;

while ( ($name, $aliases, $addrtype, $net) = getnetent() ){

    print "Name = $name\n";
    print "Aliases = $aliases\n";
    print "Addrtype = $addrtype\n";
    print "Net = $net\n";
}

setnetent(1); # Rewind the database;

while ( ($name, $aliases, $addrtype, $net) = getnetent() ){

    print "Name = $name\n";
    print "Aliases = $aliases\n";
    print "Addrtype = $addrtype\n";
    print "Net = $net\n";
}

endnetent(); # Closes the database;
```

When above code is executed, it produces the following result:
setpgroup

Description

This function sets the current process group for the process PID. You can use a value of 0 for PID to change the process group of the current process. If both arguments are omitted, defaults to values of 0. Causes a fatal error if the system does not support the function.
**Syntax**

Following is the simple syntax for this function:

```
setpgid PID, PGRP
```

**Return Value**

This function returns undef on failure and new parent process ID.

**Example**

Following is the example code showing its basic usage:

```
When above code is executed, it produces the following result:
```

---

**setpriority**

**Description**

This function sets the priority for a process (PRIO_PROCESS), process group (PRIO_PGRP), or user (PRIO_USER). The argument WHICH specifies what entity to set the priority for, and WHO is the process ID or user ID to set. A value of 0 for WHO defines the current process, process group, or user. Produces a fatal error on systems that don't support the system setpriority( ) function.

The priority is a number representing the level of priority (normally in the range 120 to 20) where the lower the priority the more favorable the scheduling of the process by the operating system.

**Syntax**

Following is the simple syntax for this function:

```
setpriority WHICH, WHO, PRIORITY
```

**Return Value**

This function does not return any value.

**Example**

Following is the example code showing its basic usage:
#!/usr/bin/perl

print("setpriority() ", setpriority(0, 0, -20), "\n");

When above code is executed, it produces the following result:

setpriority() 0

**setprotoent**

**Description**

This function should be called before the first call to getprotoent. The STAYOPEN argument is optional and unused on most systems. As getprotoent() retrieves the information for the next line in the protocol database, then setprotoent sets (or resets) the enumeration to the beginning of the set of host entries.

**Syntax**

Following is the simple syntax for this function:

setprotoent STAYOPEN

**Return Value**

This function does not return any value.

**Example**

Following is the example code showing its basic usage:

#!/usr/bin/perl

while(($name, $aliases, $protocol_number) = getprotoent()){
    print "Name = $name\n";
    print "Aliases = $aliases\n";
    print "Protocol Number = $protocol_number\n";
}

setprotoent(1); # Rewind the database.
while(($name, $aliases, $protocol_number) = getprotoent()){
  print "Name = $name\n";
  print "Aliases = $aliases\n";
  print "Protocol Number = $protocol_number\n";
}
endprotoent();  # Closes the database

When above code is executed, it produces the following result:

Name = ip
Aliases = IP
Protocol Number = 0
Name = hopopt
Aliases = HOPOPT
Protocol Number = 0
Name = icmp
Aliases = ICMP
Protocol Number = 1
Name = igmp
Aliases = IGMP
Protocol Number = 2
Name = ggp
Aliases = GGP
Protocol Number = 3
Name = ipencap
Aliases = IP-ENCAP
Protocol Number = 4
Name = st
Aliases = ST
Protocol Number = 5
.
.
.
Name = rsvp-e2e-ignore
Aliases = RSVP-E2E-IGNORE
**setpwent**

**Description**
This function Sets (or resets) the enumeration to the beginning of the set of password entries. This function should be called before the first call to getpwent.

**Syntax**
Following is the simple syntax for this function:

```perl
setpwent
```

**Return Value**
This function does not return any value.

**Example**
Following is the example code showing its basic usage:

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

---

Protocol Number = 134
Name = udplite
Aliases = UDPLite

Protocol Number = 136
Name = mpls-in-ip
Aliases = MPLS-in-IP

Protocol Number = 137
Name = manet
Aliases = manet

Protocol Number = 138
Name = hip
Aliases = HIP

Protocol Number = 139
Name = shim6
Aliases = Shim6

Protocol Number = 140
while(($name, $passwd, $uid, $gid, $quota, $comment, $gcos, $dir, $shell) = getpwent()){
    print "Name = $name\n";
    print "Password = $passwd\n";
    print "UID = $uid\n";
    print "GID = $gid\n";
    print "Quota = $quota\n";
    print "Comment = $comment\n";
    print "Gcos = $gcos\n";
    print "HOME DIR = $dir\n";
    print "Shell = $shell\n";
}

setpwent(); # Rewind the database /etc/passwd

while(($name, $passwd, $uid, $gid, $quota, $comment, $gcos, $dir, $shell) = getpwent()){
    print "Name = $name\n";
    print "Password = $passwd\n";
    print "UID = $uid\n";
    print "GID = $gid\n";
    print "Quota = $quota\n";
    print "Comment = $comment\n";
    print "Gcos = $gcos\n";
    print "HOME DIR = $dir\n";
    print "Shell = $shell\n";
}

endpwent(); # Closes the database;

When above code is executed, it produces the following result:

Name = root
Password = x
UID = 0
GID = 0
Quota =
Comment =
Gcos = root
HOME DIR = /root
Shell = /bin/bash
Name = bin
Password = x
UID = 1
GID = 1
Quota =
Comment =
Gcos = bin
HOME DIR = /bin
Shell = /sbin/nologin
Name = daemon
Password = x
UID = 2
GID = 2
Quota =
Comment =
Gcos = daemon
HOME DIR = /sbin
Shell = /sbin/nologin
.
.
.
Name = qemu
Password = x
UID = 107
GID = 107
Quota =
setservent

Description
This function should be called before the first call to getservent. The STAYOPEN argument is optional and unused on most systems. As getservent() retrieves the information for the next line in the services database, then setservent sets (or resets) the enumeration to the beginning of the set of host entries.

Syntax
Following is the simple syntax for this function:
### Return Value

This function does not return any value.

### Example

Following is the example code showing its basic usage:

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

while(($name, $aliases, $port_number, $protocol_name) = getservent()){
    print "Name = $name\n";
    print "Aliases = $aliases\n";
    print "Port Number = $port_number\n";
    print "Protocol Name = $protocol_name\n";
}

setservent();  # Rewind the database /etc/services;

while(($name, $aliases, $port_number, $protocol_name) = getservent()){
    print "Name = $name\n";
    print "Aliases = $aliases\n";
    print "Port Number = $port_number\n";
    print "Protocol Name = $protocol_name\n";
}

endservent();  # Closes the database;
```
When above code is executed, it produces the following result:

```

```

**setsockopt**

**Description**

This function Sets the socket option OPTNAME with a value of OPTVAL on SOCKET at the specified LEVEL. You will need to import the Socket module for the valid values for OPTNAME shown below in Table.

**Syntax**

Following is the simple syntax for this function:

```
setsockopt SOCKET, LEVEL, OPTNAME, OPTVAL
```

**Return Value**

This function returns undef on failure and 1 on success.

<table>
<thead>
<tr>
<th>OPTNAME</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SO_DEBUG</td>
<td>Enable/disable recording of debugging information.</td>
</tr>
<tr>
<td>SO_REUSEADDR</td>
<td>Enable/disable local address reuse.</td>
</tr>
<tr>
<td>SO_KEEPALIVE</td>
<td>Enable/disable keep connections alive.</td>
</tr>
<tr>
<td>SO_DONTROUTE</td>
<td>Enable/disable routing bypass for outgoing messages.</td>
</tr>
<tr>
<td>SO_LINGER</td>
<td>Linger on close if data is present.</td>
</tr>
<tr>
<td>SO_BROADCAST</td>
<td>Enable/disable permission to transmit broadcast messages.</td>
</tr>
<tr>
<td>SO_OOBINLINE</td>
<td>Enable/disable reception of out-of-band data in band.</td>
</tr>
<tr>
<td>SO_SNDBUF</td>
<td>Set buffer size for output.</td>
</tr>
<tr>
<td>SO_RCVBUF</td>
<td>Set buffer size for input.</td>
</tr>
<tr>
<td>SO&gt;Type</td>
<td>Get the type of the socket (get only).</td>
</tr>
<tr>
<td>SO_ERROR</td>
<td>Get and clear error on the socket (get only).</td>
</tr>
</tbody>
</table>

**Example**

Following is the example code showing its basic usage:
When above code is executed, it produces the following result:

```
1 - 2 - 3 - 4 - 5 - The End
```

### shift

#### Description
This function returns the first value in an array, deleting it and shifting the elements of the array list to the left by one. If ARRAY is not specified, shifts the @_ array within a subroutine, or @ARGV otherwise. shift is essentially identical to pop, except values are taken from the start of the array instead of the end.

#### Syntax
Following is the simple syntax for this function:

```
shift ([ARRAY])
```

#### Return Value
This function returns undef if the array is empty else returns first element in array.

#### Example
Following is the example code showing its basic usage:

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

@array = (1..5);
while ($element = shift(@array)) {
    print "$element - ";
}
print "The End\n";
```

When above code is executed, it produces the following result:

```
1 - 2 - 3 - 4 - 5 - The End
```
shmctl

Description
This function controls the shared memory segment referred to by ID, using CMD with ARG. You will need to import the IPC::SysV module to get the command tokens defined below in Table.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPC_STAT</td>
<td>Places the current value of each member of the data structure associated with ID into the scalar ARG</td>
</tr>
<tr>
<td>IPC_SET</td>
<td>Sets the value of the following members of the data structure associated with ID to the corresponding values found in the packed scalar ARG</td>
</tr>
<tr>
<td>IPC_RMID</td>
<td>Removes the shared memory identifier specified by ID from the system and destroys the shared memory segment and data structure associated with it</td>
</tr>
<tr>
<td>SHM_LOCK</td>
<td>Locks the shared memory segment specified by ID in memory</td>
</tr>
<tr>
<td>SHM_UNLOCK</td>
<td>Unlocks the shared memory segment specified by ID</td>
</tr>
</tbody>
</table>

Syntax
Following is the simple syntax for this function:

```
shmctl ID, CMD, ARG
```

Return Value
This function returns undef on failure and 0 but true if the return value from the shmctl() is 0.
Example
Following is the example code showing its basic usage:

```
#!/usr/bin/perl

# Assume this file name is writer.pl

use IPC::SysV;

#use these next two lines if the previous use fails.
eval 'sub IPC_CREAT {0001000}' unless defined &IPC_CREAT;
eval 'sub IPC_RMID {0}' unless defined &IPC_RMID;

$key = 12345;
,size = 80;
$message = "Pennyfarthingale."

# Create the shared memory segment

$key = shmget($key, $size, &IPC_CREAT | 0777 ) or
    die "Can't shmget: $!";

# Place a string in it
shmwrite( $id, $message, 0, 80 ) or die "Can't shmwrite: $!";

sleep 20;

# Delete it;
shmctl( $id, &OPC_RMID, 0 ) or die "Can't shmctl: $! ";
```

Write a reader program which retrieves the memory segment corresponding to $key and reads its contents using shmread();.

```
#!/usr/bin/perl
```
# Assume this file name is reader.pl

$key = 12345;
<size = 80;

# Identify the shared memory segment
$id = shmget( $key, $size, 0777 ) or die "Can't shmget: $!";

# Read its contents into a string
shmread($id, $var, 0, $size) or die "Can't shmread: $!";

print $var;

Now First run writer.pl program in background and then reader.pl then it will produces the following result.

$perl writer.pl&
$perl reader.pl

Pennyfrathingale

## shmget

### Description

This function returns the shared memory segment ID for the segment matching KEY. A new shared memory segment is created of at least SIZE bytes, providing that either KEY does not already have a segment associated with it or that KEY is equal to the constant IPC_PRIVATE.

### Syntax

Following is the simple syntax for this function:

shmget KEY, SIZE, FLAGS

shmget KEY
Return Value
This function returns undef on failure and Shared memory ID on success.

Example
Following is the example code showing its basic usage:

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

# Assume this file name is writer.pl

use IPC::SysV;

#use these next two lines if the previous use fails.
eval 'sub IPC_CREAT {0001000}' unless defined &IPC_CREAT;
eval 'sub IPC_RMID {0}' unless defined &IPC_RMID;

$key = 12345;
$size = 80;
$message = "Pennyfarthingale.";

# Create the shared memory segment

$key = shmget($key, $size, &IPC_CREAT | 0777 ) or die "Can't shmget: $!";

# Place a string in it
shmwrite( $id, $message, 0, 80 ) or die "Can't shmwrite: $!";

sleep 20;

# Delete it;
shmctl( $id, &OPC_RMID, 0 ) or die "Can't shmctl: $! ";
```
Write a reader program which retrieves the memory segment corresponding to $key and reads its contents using shmread();.

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

# Assume this file name is reader.pl
$key = 12345;
$size = 80;

# Identify the shared memory segment
$id = shmget( $key, $size, 0777 ) or die "Can't shmget: $!";

# Read its contents into a string
shmread($id, $var, 0, $size) or die "Can't shmread: $!";

print $var;
```

Now First run writer.pl program in background and then reader.pl then it will produces the following result.

```bash
$perl writer.pl&
$perl reader.pl

Pennyfrathingale
```

**shmread**

**Description**

This function reads the shared memory segment ID into the scalar VAR at position POS for up to SIZE bytes.

**Syntax**

Following is the simple syntax for this function:

```
shmread ID, VAR, POS, SIZE
```
**Return Value**

This function returns 0 on failure and 1 on success.

**Example**

Following is the example code showing its basic usage:

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

# Assume this file name is writer.pl

use IPC::SysV;

#use these next two lines if the previous use fails.
eval 'sub IPC_CREAT {0001000}' unless defined &IPC_CREAT;
eval 'sub IPC_RMID {0}' unless defined &IPC_RMID;

$key = 12345;
,size = 80;
$message = "Pennyfarthingale.";

# Create the shared memory segment

$key = shmget($key, $size, &IPC_CREAT | 0777 ) or
die "Can't shmget: $!";

# Place a string in it
shmwrite( $id, $message, 0, 80 ) or die "Can't shmwrite: $!";

sleep 20;

# Delete it;
shmctl( $id, &OPC_RMD, 0 ) or die "Can't shmctl: $! ";
```
Write a reader program which retrieves the memory segment corresponding to
$key and reads its contents using shmread();.

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

# Assume this file name is reader.pl

$key = 12345;
$size = 80;

# Identify the shared memory segment
$id = shmget($key, $size, 0777) or die "Can't shmget: $!";

# Read its contents into a string
shmread($id, $var, 0, $size) or die "Can't shmread: $!";

print $var;
```

Now First run writer.pl program in background and then reader.pl then it will
produces the following result.

```
$ perl writer.pl&
$ perl reader.pl

Pennyfrathingale
```

**shmwrite**

**Description**

This function writes STRING from the position POS for SIZE bytes into the
shared memory segment specified by ID. The SIZE is greater than the length of
STRING. shmwrite appends null bytes to fill out to SIZE bytes.

**Syntax**

Following is the simple syntax for this function:

```
shmwrite ID, STRING, POS, SIZE
```
**Return Value**
This function returns 0 on failure and 1 on success.

**Example**
Following is the example code showing its basic usage:

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

# Assume this file name is writer.pl

use IPC::SysV;

#use these next two lines if the previous use fails.
eval 'sub IPC_CREAT {0001000}' unless defined &IPC_CREAT;
eval 'sub IPC_RMID {0}' unless defined &IPC_RMID;

$key  = 12345;
$size = 80;
$message = "Pennyfarthingale.";

# Create the shared memory segment

$key = shmget($key, $size, &IPC_CREAT | 0777 ) or die "Can't shmget: $!";

# Place a string in it
shmwrite( $id, $message, 0, 80 ) or die "Can't shmwrite: $!";

sleep 20;

# Delete it;
shmctl( $id, &IPC_RMID, 0 ) or die "Can't shmctl: $! ";
```

Write a reader program which retrieves the memory segment corresponding to $key and reads its contents using shmread();.
#!/usr/bin/perl

# Assume this file name is reader.pl

$key = 12345;
$size = 80;

# Identify the shared memory segment
$id = shmget($key, $size, 0777 ) or die "Can't shmget: $!";

# Read its contents into a string
shmread($id, $var, 0, $size) or die "Can't shmread: $!";

print $var;

Now First run writer.pl program in background and then reader.pl then it will produces the following result.

$ perl writer.pl &
$ perl reader.pl

Pennyfrathingale

shutdown

Description

This function disables a socket connection according to the value of HOW. The valid values for HOW are identical to the system call of the same name. A value of 0 indicates that you have stopped reading information from the socket.

A value of 1 indicates that you have stopped writing to the socket. A value of 2 indicates that you have stopped using the socket altogether.

Syntax

Following is the simple syntax for this function:

shutdown SOCKET, HOW
**Return Value**
This function returns 0 on failure and 1 on success.

**Example**
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl
$temp = sin(4);
print "sin value of 4 is $temp";
```

When above code is executed, it produces the following result:

**sin**

**Description**
This function returns the sine of EXPR, or $._ if not specified. This function always returns a floating point.

**Syntax**
Following is the simple syntax for this function:

```perl
sin EXPR
```

**Return Value**
This function returns the Floating Point sin value of EXPR.

**Example**
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl
$temp = sin(4);
print "sin value of 4 is $temp";
```

When above code is executed, it produces the following result:
sin value of 4 is -0.756802495307928

sleep

Description
This function Pauses the script for EXPR seconds, or forever if EXPR is not specified. Returns the number of seconds actually slept. Can be interrupted by a signal handler, but you should avoid using sleep with alarm, since many systems use alarm for the sleep implementation.

Syntax
Following is the simple syntax for this function:

```perl
sleep EXPR
```

Return Value
This function returns Integer, number of seconds actually slept

Example
Following is the example code showing its basic usage:

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

$num = 5;
while($num--){
    sleep(1);
}
```

When above code is executed, it produces the following result:
What is a Socket?

Socket is a Berkeley UNIX mechanism of creating a virtual duplex connection between different processes. This was later ported on to every known OS enabling communication between systems across geographical location running on different OS software. If not for the socket, most of the network communication between systems would never ever have happened.

Taking a closer look; a typical computer system on a network receives and sends information as desired by the various applications running on it. This information is routed to the system, since a unique IP address is designated to it. On the system, this information is given to the relevant applications which listen on different ports. For example an internet browser listens on port 80 for information received from the web server. Also we can write our custom applications which may listen and send/receive information on a specific port number.

For now, let's sum up that a socket is an IP address and a port, enabling connection to send and recieve data over a network.

To explain above mentioned socket concept we will take an example of Client - Server Programming using Perl. To complete a client server architecture we would have to go through the following steps:

**To create a server**

- Create a socket using `socket` call.
- Bind the socket to a port address using `bind` call.
- Listen to the socket at the port address using `listen` call.
- Accept client connections using `accept` call.

**To create a client**

- Create a socket with `socket` call.
- Connect (the socket) to the server using `connect` call.

Following diagram shows complete sequence of the calls used by Client and Server to communicate with each other:
Server Side Socket Calls

The socket() call

The `socket()` call is the first call in establishing a network connection is creating a socket. This call has following syntax:

```perl
socket( SOCKET, DOMAIN, TYPE, PROTOCOL );
```

Above call creates a SOCKET and other three arguments are integers which should have the following values for TCP/IP connections.

- **DOMAIN** should be PF_INET. It's probable 2 on your computer.
- **TYPE** should be SOCK_STREAM for TCP/IP connection.
- **PROTOCOL** should be `(getprotobyname('tcp'))[2]`. It is the particular protocol such as TCP to be spoken over the socket.

So socket function call issued by the server will be something like this:

```perl
use Socket  # This defines PF_INET and SOCK_STREAM

socket(SOCKET,PF_INET,SOCK_STREAM,(getprotobyname('tcp'))[2]);
```
The **bind() call**

The sockets created by socket() call are useless until they are bound to a hostname and a port number. Server uses following **bind()** function to specify the port at which they will be accepting connections from the clients.

```perl
bind( SOCKET, ADDRESS );
```

Here **SOCKET** is the descriptor returned by socket() call and **ADDRESS** is a socket address (for TCP/IP) containing three elements:

- The address family (For TCP/IP, that's AF_INET, probably 2 on your system)
- The port number (for example 21)
- The internet address of the computer (for example 10.12.12.168)

As the bind() is used by a server which does not need to know its own address so the argument list looks like this:

```perl
use Socket # This defines PF_INET and SOCK_STREAM

$port = 12345; # The unique port used by the server to listen requests
$server_ip_address = "10.12.12.168";
bind( SOCKET, pack_sockaddr_in($port, inet_aton($server_ip_address)))
    or die "Can't bind to port $port! \n";
```

The **or die** clause is very important because if a server dies without outstanding connections the port won't be immediately reusable unless you use the option SO_REUSEADDR using **setsockopt()** function. Here **pack_sockaddr_in()** function is being used to pack the Port and IP address into binary format.

The **listen() call**

If this is a server program then it is required to issue a call to **listen()** on the specified port to listen ie wait for the incoming requests. This call has following syntax:

```perl
listen( SOCKET, QUEUESIZE );
```

The above call uses **SOCKET** descriptor returned by socket() call and **QUEUESIZE** is the maximum number of outstanding connection request allowed simultaneously.
The accept() call

If this is a server program then it is required to issue a call to access() function to accept the incoming connections. This call has following syntax:

```perl
accept( NEW_SOCKET, SOCKET );
```

The accept call receive SOCKET descriptor returned by socket() function and upon successful completion, a new socket descriptor NEW_SOCKET is returned for all future communication between client and server. If access() call fails then it returns FLASE which is defined in Socket module which we have used initially.

Generally, accept() is used in an infinite loop. As soon as one connection arrives the server either creates a child process to deal with it or serves it himself and then goes back to listen for more connections.

```perl
while(1) {
    accept( NEW_SOCKET, SOCKET );
    .......
}
```

Now all the calls related to server are over and let us see a call which will be required by the client

Client Side Socket Calls

The connect() call

If you are going to prepare client program then first you will use socket() call to create a socket and then you would have to use connect() call to connect to the server. You already have seen socket() call syntax and it will remain similar to server socket() call, but here is the syntax for connect() call:

```perl
connect( SOCKET, ADDRESS );
```

Here SCOKET is the socket descriptor returned by socket() call issued by the client and ADDRESS is a socket address similar to bind call, except that it contains the IP address of the remote server.

```perl
$port = 21;       # For example, the ftp port
$server_ip_address = "10.12.12.168";
connect( SOCKET, pack_sockaddr_in($port, inet_aton($server_ip_address)))
    or die "Can't connect to port $port! \n";
```
If you connect to the server successfully then you can start sending your commands to the server using SOCKET descriptor, otherwise your client will come out by giving an error message.

**Client - Server Example**

Following is a Perl code to implement a simple client-server program using Perl socket. Here server listens for incoming requests and once connection is established, it simply replies *Smile from the server*. The client reads that message and print on the screen. Let's see how it has been done, assuming we have our server and client on the same machine.

**Script to Create a Server**

```perl
#!/usr/bin/perl -w
#
# Filename : server.pl

use strict;
use Socket;

# use port 7890 as default
my $port = shift || 7890;
my $proto = getprotobynumber('tcp');
my $server = "localhost"; # Host IP running the server

# create a socket, make it reusable
socket(SOCKET, PF_INET, SOCK_STREAM, $proto)
   or die "Can't open socket $!\n";
setsockopt(SOCKET, SOL_SOCKET, SO_REUSEADDR, 1)
   or die "Can't set socket option to SO_REUSEADDR $!\n";

# bind to a port, then listen
bind( SOCKET, pack_sockaddr_in($port, inet_aton($server)))
   or die "Can't bind to port $port! \n";

listen(SOCKET, 5) or die "listen: $!";
print "SERVER started on port $port\n";
```
# accepting a connection
my $client_addr;
while ($client_addr = accept(NEW_SOCKET, SOCKET)) {
    # send them a message, close connection
    my $name = gethostbyaddr($client_addr, AF_INET);
    print NEW_SOCKET "Smile from the server";
    print "Connection recievied from $name\n";
    close NEW_SOCKET;
}

To run the server in background mode issue the following command on Unix prompt:

perl sever.pl&

Script to Create a Client

#!/usr/bin/perl -w
# Filename : client.pl

use strict;
use Socket;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 7890;
my $server = "localhost"; # Host IP running the server

# create the socket, connect to the port
socket(SOCKET,PF_INET,SOCK_STREAM,(getprotobynname('tcp'))[2])
    or die "Can't create a socket $!\n";
connect( SOCKET, pack_sockaddr_in($port, inet_aton($server)))
    or die "Can't connect to port $port! \n";
my $line;
while ($line = <SOCKET>) {
    print "$line\n";
}
close SOCKET or die "close: "$!;

Now let's start our client at the command prompt which will connect to the server and read message sent by the server and displays the same on the screen as follows:

```bash
$ perl client.pl
Smile from the server
```

**NOTE:** If you are giving actual IP address in dot notation then it is recommended to provide IP address in the same format in both client as well as server to avoid any confusion.

---

### socketpair

**Description**

This function creates an unnamed pair of connected sockets in the specified DOMAIN, of the specified TYPE, using PROTOCOL. If the system socketpair( ) function is not implemented, then it causes a fatal error.

**Syntax**

Following is the simple syntax for this function:

```
socketpair SOCKET1, SOCKET2, DOMAIN, TYPE, PROTOCOL
```

**Return Value**

This function returns 0 on failure ans 1 on success.

**Example**

Following is the example code showing its basic usage:

```perl

When above code is executed, it produces the following result:
```
**sort**

**Description**

This function sorts LIST according to the subroutine SUBNAME or the anonymous subroutine specified by BLOCK. If no SUBNAME or BLOCK is specified, then it sorts according to normal alphabetical sequence.

If BLOCK or SUBNAME is specified, then the subroutine should return an integer less than, greater than, or equal to zero, according to how the elements of the array are to be sorted.

**Syntax**

Following is the simple syntax for this function:

```
sort SUBNAME LIST
sort BLOCK LIST
sort LIST
```

**Return Value**

This function returns sorted list.

**Example**

Following is the example code showing its basic usage:

```
#!/usr/bin/perl -w

$array = ("z", "w", "r", "i", "b", "a");
print("sort() ", sort(@array), "\n");
```

When above code is executed, it produces the following result:

```
sort() abirwz
```
splice

Description
This function removes the elements of ARRAY from the element OFFSET for LENGTH elements, replacing the elements removed with LIST, if specified. If LENGTH is omitted, removes everything from OFFSET onwards.

Syntax
Following is the simple syntax for this function:

```
splice ARRAY, OFFSET, LENGTH, LIST
splice ARRAY, OFFSET, LENGTH
splice ARRAY, OFFSET
```

Return Value
This function returns:

- In scalar context undef if no elements removed
- In scalar context last element removed
- In list context empty list on failure
- In list context list of elements removed

Example
Following is the example code showing its basic usage:

```
#!/usr/bin/perl -w

$array = ("a", "e", "i", "o", "u");
@removedItems = splice(@array, 0, 3, ("A", "E", "I"));

print "Removed items: @removedItems\n";
```

When above code is executed, it produces the following result:

```
Removed items: a e i
```
split

Description
This function splits a string expression into fields based on the delimiter specified by PATTERN. If no pattern is specified whitespace is the default. An optional limit restricts the number of elements returned.

A negative limit has the same effect as no limit. This function is often used in conjunction with join() to create small text databases.

Syntax
Following is the simple syntax for this function:

```perl
split /PATTERN/, EXPR, LIMIT
split /PATTERN/, EXPR
split /PATTERN/
split
```

Return Value
- Return Value in Scalar Context: Not recommended, but it returns the number of fields found and stored the fields in the @_ array.
- Return Value in Array Context: A list of fields found in EXPR or $_ if no expression is specified.

Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

@fields = split(/:/, "1:2:3:4:5");
print "Field values are: @fields\n";
```

When above code is executed, it produces the following result:

Field values are: 1 2 3 4 5
## sprintf

### Description
This function uses FORMAT to return a formatted string based on the values in LIST. Essentially identical to printf, but the formatted string is returned instead of being printed.

### Syntax
Following is the simple syntax for this function:

```plaintext
sprintf FORMAT, LIST
```

### Return Value
This function returns SCALAR, a formatted text string.

### Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

$text = sprintf("%0d \n", 9);

print "Formated string $text\n";
```

When above code is executed, it produces the following result:

```
Formated string 9
```

## sqrt

### Description
This function returns the square root of EXPR, or $_ if omitted. Most of the time, this function returns a floating point number.

### Syntax
Following is the simple syntax for this function:

```plaintext
sqrt EXPR
```
### sqrt

#### Return Value
This function returns the Floating point number.

#### Example
Following is the example code showing its basic usage:

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

$result = sqrt(4);
print " sqrt(4) is $result
"
```

When above code is executed, it produces the following result:

```
sqrt(4) is 2
```

### srand

#### Description
This function sets the seed value for the random number generator to EXPR or to a random value based on the time, process ID, and other values if EXPR is omitted.

#### Syntax
Following is the simple syntax for this function:

```perl
srand EXPR
srand
```

#### Return Value
This function does not return any value.

#### Example
Following is the example code showing its basic usage:
#!/usr/bin/perl -w

srand(26);
print("Here's a random number: ", rand(), ".\n");
srand(26);
print("Here's the same random number: ", rand(), ".\n");

When above code is executed, it produces the following result:

| Here's a random number: | 0.81688061411591. |
| Here's the same random number: | 0.81688061411591. |

### stat

**Description**

This function returns a 13-element array giving the status info for a file, specified by either FILEHANDLE, EXPR, or $_. The list of values returned is shown below in Table. If used in a scalar context, returns 0 on failure, 1 on success.

Note that support for some of these elements is system dependent. Check the documentation for a complete list.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Device number of file system</td>
</tr>
<tr>
<td>1</td>
<td>Inode number</td>
</tr>
<tr>
<td>2</td>
<td>File mode (type and permissions)</td>
</tr>
<tr>
<td>3</td>
<td>Number of (hard) links to the file</td>
</tr>
<tr>
<td>4</td>
<td>Numeric user ID of file's owner</td>
</tr>
<tr>
<td>5</td>
<td>Numeric group ID of file's owner</td>
</tr>
<tr>
<td>6</td>
<td>The device identifier (special files only)</td>
</tr>
<tr>
<td>7</td>
<td>File size, in bytes</td>
</tr>
<tr>
<td>8</td>
<td>Last access time since the epoch</td>
</tr>
<tr>
<td>9</td>
<td>Last modify time since the epoch</td>
</tr>
<tr>
<td>10</td>
<td>Inode change time (not creation time!) since the epoch</td>
</tr>
<tr>
<td>11</td>
<td>Preferred block size for file system I/O</td>
</tr>
<tr>
<td>12</td>
<td>Actual number of blocks allocated</td>
</tr>
</tbody>
</table>
**Syntax**

Following is the simple syntax for this function:

```
stat FILEHANDLE
stat EXPR
stat
```

**Return Value**

This function returns ARRAY, ($device, $inode, $mode, $nlink, $uid, $gid, $rdev, $size, $atime, $mtime, $ctime, $blksize, $blocks)

**Example**

Following is the example code showing its basic usage:

```
#!/usr/bin/perl -w

($device, $inode, $mode, $nlink, $uid, $gid, $rdev, $size,
 $atime, $mtime, $ctime, $blksize, $blocks) =
  stat("/etc/passwd");

print("stat() $device, $inode, $ctime
");
```

When above code is executed, it produces the following result:

```
stat() 147, 20212116, 1177094582
```

**study**

**Description**

This function takes extra time to study EXPR in order to improve the performance on regular expressions conducted on EXPR. If EXPR is omitted, uses $_. The actual speed gains may be very small, depending on the number of times you expect to search the string.

You can only study one expression or scalar at any one time.
**Syntax**

Following is the simple syntax for this function:

```perl
study EXPR
```

**Return Value**

This function does not return any value.

**Example**

Following is the example code showing its basic usage:

```perl

```

When above code is executed, it produces the following result:

```perl

```

**sub**

**Description**

This function defines a new subroutine. The arguments shown above follow these rules:

- **NAME** is the name of the subroutine. Named subroutines can be predeclared (without an associated code block) with, or without, prototype specifications.
- Anonymous subroutines must have a definition.
- **PROTO** defines the prototype for a function, which will be used when the function is called to validate the supplied arguments.
- **ATTRS** define additional information for the parser about the subroutine being declared.

**Syntax**

Following is the simple syntax for this function:

```perl
sub NAME PROTO ATTRS BLOCK# Named, prototype, attributes, definition
sub NAME ATTRS BLOCK # Named, attributes, definition
```
Return Value
This function does not return any value.

Example
Following is the example code showing its basic usage:

```
```

When above code is executed, it produces the following result:

```
```

substr

Description
This function returns a substring of EXPR, starting at OFFSET within the string. If OFFSET is negative, starts that many characters from the end of the string. If LEN is specified, returns that number of bytes, or all bytes up until end-of-string if not specified. If LEN is negative, leaves that many characters off the end of the string.

If REPLACEMENT is specified, replaces the substring with the REPLACEMENT string.

If you specify a substring that passes beyond the end of the string, it returns only the valid element of the original string.
**Syntax**

Following is the simple syntax for this function:

```perl
substr EXPR, OFFSET, LEN, REPLACEMENT
substr EXPR, OFFSET, LEN
substr EXPR, OFFSET
```

**Return Value**

This function returns string.

**Example**

Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

$temp = substr("okay", 2);
print "Substring value is $temp\n";

$temp = substr("okay", 1,2);
print "Substring value is $temp\n";
```

When above code is executed, it produces the following result:

- Substring value is ay
- Substring value is ka

---

**symlink**

**Description**

This function creates a symbolic link between OLDFILE and NEWFILE. On systems that don't support symbolic links, causes a fatal error.

**Syntax**

Following is the simple syntax for this function:
symlink ( OLDFILE, NEWFILE )

**Return Value**

This function returns 0 on failure and 1 on success.

**Example**

Following is the example code showing its basic usage, first create one file test.txt in /tmp directory and then try out following example it will create a symbolic link in the same directory:

```perl
#!/usr/bin/perl -w

symlink("/tmp/text.txt", "/tmp/symlink_to_text.txt");
```

When above code is executed, it produces the following result:

Symbolic link gets created

---

**syscall**

**Description**

This function calls the system call specified as the first element of the list, passing the remaining elements as arguments to the system call. If a given argument is numeric, the argument is passed as an int. If not, the pointer to the string value is passed.

**Syntax**

Following is the simple syntax for this function:

```perl
syscall EXPR, LIST
```

**Return Value**

This function returns -1 on failure of system call and values returned by system function on success.

**Example**

Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w
```
require("syscall.ph");
$pid = syscall(SYS_getpid);

print "PID of this process is $pid\n";

# To create directory use the following
$string = "newdir";
syscall(SYS_mkdir, $string);

When above code is executed, it produces the following result:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O_RDONLY</td>
<td>Read only.</td>
</tr>
<tr>
<td>O_WRONLY</td>
<td>Write only.</td>
</tr>
<tr>
<td>O_RDWR</td>
<td>Read and write.</td>
</tr>
<tr>
<td>O_CREAT</td>
<td>Create the file if it doesn’t already exist.</td>
</tr>
<tr>
<td>O_EXCL</td>
<td>Fail if the file already exists.</td>
</tr>
</tbody>
</table>
Perl

Syntax
Following is the simple syntax for this function:

```plaintext
sysopen FILEHANDLE, FILENAME, MODE, PERMS

sysopen FILEHANDLE, FILENAME, MODE
```

Return Value
This function returns 0 on failure and 1 on success.

Example
Following is the example code showing its basic usage:

When above code is executed, it produces the following result:
sysread

Description
This function is the equivalent of the C/operating system function read() because it bypasses the buffering system employed by functions like print, read, and seek, it should only be used with the corresponding syswrite and sysseek functions.

It read LENGTH bytes from FILEHANDLE, placing the result in SCALAR. If OFFSET is specified, then data is written to SCALAR from OFFSET bytes, effectively appending the information from a specific point. If OFFSET is negative, it starts from the number of bytes specified counted backward from the end of the string.

Syntax
Following is the simple syntax for this function:

```
sysread FILEHANDLE, SCALAR, LENGTH, OFFSET
sysread FILEHANDLE, SCALAR, LENGTH
```

Return Value
This function returns undef on error, 0 at end of file and Integer, number of bytes read on success.

Example
Following is the example code showing its basic usage:

```
```

When above code is executed, it produces the following result:

```
```

sysseek

Description
This function sets the position within FILEHANDLE according to the values of POSITION and WHENCE.
This is equivalent of the C function lseek( ), so you should avoid using it with buffered forms of FILEHANDLE. This includes the "FILEHANDLE" notation and print, write, seek, and tell. Using it with sysread or syswrite is OK, since they too ignore buffering.

The position within the file is specified by POSITION, using the value of WHENCE as a reference point, as shown below in Table.

<table>
<thead>
<tr>
<th>SEEK_SET</th>
<th>SEEK_CUR</th>
<th>SEEK_END</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rightarrow 0 )</td>
<td>( \rightarrow 1 )</td>
<td>( \rightarrow 2 )</td>
</tr>
</tbody>
</table>

Sets the new position absolutely to \( \text{POSITION} \) bytes within the file

Sets the new position to the current position plus \( \text{POSITION} \) bytes within the file

Sets the new position to \( \text{POSITION} \) bytes, relative to the end of the file

**Syntax**

Following is the simple syntax for this function:

```
sysseek FILEHANDLE, POSITION, WHENCE
```

**Return Value**

This function returns undef on failure, a position of 0 is returned as the string 0 but true and Integer, new position (in bytes) on success.

**Example**

Following is the example code showing its basic usage:

When above code is executed, it produces the following result:
system

Description
This function executes the command specified by PROGRAM, passing LIST as arguments to the command.

The return value is the exit status of the program as returned by the wait function. To obtain the actual exit value, divide by 256.

Syntax
Following is the simple syntax for this function:

```
system PROGRAM, LIST

system PROGRAM
```

Return Value
This function returns exit status of program as returned by wait.

Example
Following is the example code showing its basic usage:

```
#!/usr/bin/perl -w

system("ls -F /var > /tmp/t.tmp");
```

When above code is executed, it produces the following result:

```
A file in /tmp directory, check it out.
```

syswrite

Description
This function attempts to write LENGTH bytes from SCALAR to the file associated with FILEHANDLE. If OFFSET is specified, then information is read from OFFSET bytes in the supplied SCALAR. This function uses the C/operating system write( ) function, which bypasses the normal buffering.
**Perl**

### Syntax
Following is the simple syntax for this function:

```perl
syswrite FILEHANDLE, SCALAR, LENGTH, OFFSET
```

```perl
syswrite FILEHANDLE, SCALAR, LENGTH
```

### Return Value
This function returns `undef` on error and `Integer`, number of bytes written on success.

### Example
Following is the example code showing its basic usage:

When above code is executed, it produces the following result:

---

**tell**

### Description
This function returns the current position of read pointer (in bytes) within the specified FILEHANDLE. If FILEHANDLE is omitted, then it returns the position within the last file accessed.

### Syntax
Following is the simple syntax for this function:

```perl
tell FILEHANDLE
```

```perl
tell
```

### Return Value
This function returns current file position in bytes.

### Example
Following is the example code showing its basic usage, to check this function do the followings:

- Create a text file with "this is test" as content and store it into /tmp directory.
- Read 2 characters from this file.
- Now check the position of read pointer in the file.

```perl
#!/usr/bin/perl
open( FILE, "<tmp/test.txt" ) || die "Unable to open test file";
$char = getc( FILE );
print "First Character is $char\n";
$char = getc( FILE );
print "Second Character is $char\n";
# Now check the position of read pointer.
$position = tell( FILE );
print "Position within file $position\n";
close(FILE);
```

When above code is executed, it produces the following result:

```
First Character is T
Second Character is h
Position within file 2
```

**telldir**

**Description**

This function returns the current position of read pointer within the directory listing referred to by DIRHANDLE. This returned value can be used by seekdir() function.

**Syntax**

Following is the simple syntax for this function:

```
telldir DIRHANDLE
```
**Return Value**
This function returns the current position within the directory.

**Example**
Following is the example code showing its basic usage, we have only two files in /tmp directory:

```perl
#!/usr/bin/perl -w
opendir(DIR, "/tmp");

print("Position without read : ", telldir(DIR), "\n");

$dir = readdir(DIR);
print("Position after one read : ", telldir(DIR), "\n");
print "$dir\n";
seekdir(DIR,0);

$dir = readdir(DIR);
print "$dir\n";
print("Position after second read : ", telldir(DIR), "\n");

closedir(DIR);
```

When above code is executed, it produces the following result:

```
Position without read : 0
Position after one read : 1220443271
test.txt
test.txt
Position after second read : 1220443271
```

**tie**

**Description**
This function ties the VARIABLE to the package class CLASSNAME that provides implementation for the variable type. Any additional arguments in LIST are
passed to the constructor for the entire class. Typically used to bind hash variables to DBM databases.

**Syntax**
Following is the simple syntax for this function:

```
tie VARIABLE, CLASSNAME, LIST
```

**Return Value**
This function returns reference to tied object.

**Example**
Following is the example code showing its basic usage, we have only two files in /tmp directory:

```perl
#!/usr/bin/perl -w

package MyArray;

sub TIEARRAY {
    print "TYING
";
    bless [];}

sub DESTROY {
    print "DESTROYING
";
}

sub STORE {
    my ($self, $index, $value ) = @_;
    print "STORING $value at index $index\n";
    $self[$index] = $value;
}

sub FETCH {
    my ($self, $index ) = @_;
    print "FETCHING the value at index $index\n";
    return $self[$index];
}
```
package main;

object = tie @x, MyArray; #@x is now a MyArray array;

print "object is a ", ref($object), "\n";

$x[0] = 'This is test'; #this will call STORE();
print $x[0], "\n"; #this will call FETCH();
print $object->FETCH(0), "\n";
untie @x #now @x is a normal array again.

When above code is executed, it produces the following result:

TYING
object is a MyArray
STORING This is test at index 0
FETCHING the value at index 0
This is test
FETCHING the value at index 0
This is test
DESTROYING

When the tie function is called, what actually happens is that the TIESCALAR method from FileOwner is called, passing '.bash_profile' as the argument to the method. This returns an object, which is associated by tie to the $profile variable.

When $profile is used in the print statements, the FETCH method is called. When you assign a value to $profile, the STORE method is called, with 'mcslp' as the argument to the method. If you can follow this, then you can create tied scalars, arrays, and hashes, since they all follow the same basic model. Now let's examine the details of our new FileOwner class, starting with the TIESCALAR method:
**tied**

**Description**
This function returns a reference to the object underlying the tied entity VARIABLE. To understand tied check tie function.

**Syntax**
Following is the simple syntax for this function:

```
tied VARIABLE
```

**Return Value**
This function returns undef if VARIABLE is not tied to a package otherwise it returns a reference to the object.

**Example**
Following is the example code showing its basic usage:

```
```
When above code is executed, it produces the following result:

```
```

**time**

**Description**
This function returns the number of seconds since the epoch (00:00:00 UTC, January 1, 1970, for most systems; 00:00:00, January 1, 1904, for Mac OS). Suitable for feeding to gmtime and localtime.

**Syntax**
Following is the simple syntax for this function:

```
time
```

**Return Value**
This function returns integer, seconds since epoch.
Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

@weekday = ("Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat");

$retval = time();

print "Return time is $retval\n";
$local_time = gmtime($retval);

print "Local time = $local_time\n";
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = gmtime(time);
$year = $year + 1900;
$mon += 1;
print "Formatted time = $mday/$mon/$year $hour:$min:$sec $weekday[$wday]\n";
```

When above code is executed, it produces the following result:

Return time is 1176831539
Local time = Tue Apr 17 17:38:59 2007
Formatted time = 17/4/2007 17:38:59 Tue

`times`

Description
This function returns a four-element list giving the user, system, child, and child system times for the current process and its children.

Syntax
Following is the simple syntax for this function:

```perl
times
```
Return Value
This function returns ARRAY, ($usertime, $systemtime, $childsystem, $childuser)

Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

($usertime, $systemtime, $childsystem, $childuser) = times();
print("times() $usertime $systemtime $childsystem $childuser\n");
```

When above code is executed, it produces the following result:

```
times() 0 0 0 0
```

tr

Description
This is not a function. This is the transliteration operator; it replaces all occurrences of the characters in SEARCHLIST with the characters in REPLACEMENTLIST.

Syntax
Following is the simple syntax for this function:

```
tr/SEARCHLIST/REPLACEMENTLIST/
```

Return Value
This function returns number of characters replaced or deleted.

Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

$string = 'the cat sat on the mat.';
$string =~ tr/a-z/b/d;
```
print "$string\n";

When above code is executed, it produces the following result. Here option d is used to delete matched characters.

```
b b b.
```

### truncate

**Description**

This function truncates (reduces) the size of the file specified by FILEHANDLE to the specified LENGTH (in bytes). Produces a fatal error if the function is not implemented on your system.

**Syntax**

Following is the simple syntax for this function:

```
truncate FILEHANDLE, LENGTH
```

**Return Value**

This function returns undef if the operation failed and 1 on success.

**Example**

Following is the example code showing its basic usage, it will truncate file "test.txt" to zero length:

```

```

When above code is executed, it produces the following result:

```

```

### uc

**Description**

This function returns an uppercased version of EXPR, or $$_ if not specified. Check ucfirst() function which will return only first character in uppercase.
Syntax
Following is the simple syntax for this function:

```
uc EXPR
```

Return Value
This function returns String in uppercase.

Example
Following is the example code showing its basic usage:

```
#!/usr/bin/perl -w

$string = 'the cat sat on the mat.';
$u_string = uc($string);

print "First String |$string|\n";
print "Second String |$u_string|\n";
```

When above code is executed, it produces the following result:

```
First String |the cat sat on the mat.|
Second String |THE CAT SAT ON THE MAT.|
```

ucfirst

Description
This function returns the value of EXPR with only the first character uppercased.
If EXPR is omitted, then uses $._.

Syntax
Following is the simple syntax for this function:

```
ucfirst EXPR
```
ucfirst

**Return Value**
This function returns String with first character in uppercase.

**Example**
Following is the example code showing its basic usage:

```
#!/usr/bin/perl -w

$string = 'the cat sat on the mat.';
$u_string = ucfirst($string);

print "First String |$string|\n";
print "Second String |$u_string|\n";
```

When above code is executed, it produces the following result:

```
First String |the cat sat on the mat.|
Second String |The cat sat on the mat.|
```

umask

**Description**
This function sets the umask (default mask applied when creating files and directories) for the current process. Value of EXPR must be an octal number. If EXPR is omitted, simply returns the previous value.

**Syntax**
Following is the simple syntax for this function:

```
umask EXPR

umask
```

**Return Value**
This function returns the previous umask value.
Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

print("The current umask is: ", umask(), "\n");
```

When above code is executed, it produces the following result. You can get different result on your computer based on your setting.

```perl
The current umask is: 18
```

**undef**

**Description**
This function undefines the value of EXPR. Use on a scalar, list, hash, function, or typeglob. Use on a hash with a statement such as `undef $hash{$key};` actually sets the value of the specified key to an undefined value.

If you want to delete the element from the hash, use the delete function.

**Syntax**
Following is the simple syntax for this function:

```perl
undef EXPR
```

**Return Value**
This function returns `undef`.

**Example**
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

$scalar = 10;
$array = (1,2);
```
print "1 - Value of Scalar is $scalar\n";
print "1 - Value of Array is @array\n";

undef( $scalar );
undef( @array );

print "2 - Value of Scalar is $scalar\n";
print "2 - Value of Array is @array\n";

When above code is executed, it produces the following result:

1 - Value of Scalar is 10
1 - Value of Array is 1 2
2 - Value of Scalar is
2 - Value of Array is
Use of uninitialized value $scalar in concatenation (.) or string at main.pl line 12.

unlink

Description
This function deletes the files specified by LIST, or the file specified by $_ otherwise. Be careful while using this function because there is no recovering once a file gets deleted.

Syntax
Following is the simple syntax for this function:

unlink LIST

unlink

Return Value
This function returns the number of files deleted.

Example
Following is the example code showing its basic usage, create two files t1.txt and t2.txt in /tmp directory and then use the following program to delete these two files:

```perl
#!/usr/bin/perl -w
unlink( "/tmp/t1.txt", "/tmp/t2.txt" );
```

When above code is executed, it produces the following result:

Both the files t1.txt and t2.txt will be deleted from /tmp.

### unpack

**Description**

This function unpacks the binary string STRING using the format specified in TEMPLATE. Basically reverses the operation of pack, returning the list of packed values according to the supplied format.

You can also prefix any format field with a %<number> to indicate that you want a 16-bit checksum of the value of STRING, instead of the value.

**Syntax**

Following is the simple syntax for this function:

```
unpack TEMPLATE, STRING
```

**Return Value**

This function returns the list of unpacked values.

Here is the table which gives values to be used in TEMPLATE.

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>ASCII character string padded with null characters</td>
</tr>
<tr>
<td>A</td>
<td>ASCII character string padded with spaces</td>
</tr>
<tr>
<td>b</td>
<td>String of bits, lowest first</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>B</td>
<td>String of bits, highest first</td>
</tr>
<tr>
<td>c</td>
<td>A signed character (range usually -128 to 127)</td>
</tr>
<tr>
<td>C</td>
<td>An unsigned character (usually 8 bits)</td>
</tr>
<tr>
<td>d</td>
<td>A double-precision floating-point number</td>
</tr>
<tr>
<td>f</td>
<td>A single-precision floating-point number</td>
</tr>
<tr>
<td>h</td>
<td>Hexadecimal string, lowest digit first</td>
</tr>
<tr>
<td>H</td>
<td>Hexadecimal string, highest digit first</td>
</tr>
<tr>
<td>i</td>
<td>A signed integer</td>
</tr>
<tr>
<td>I</td>
<td>An unsigned integer</td>
</tr>
<tr>
<td>l</td>
<td>A signed long integer</td>
</tr>
<tr>
<td>L</td>
<td>An unsigned long integer</td>
</tr>
<tr>
<td>n</td>
<td>A short integer in network order</td>
</tr>
<tr>
<td>N</td>
<td>A long integer in network order</td>
</tr>
<tr>
<td>p</td>
<td>A pointer to a string</td>
</tr>
<tr>
<td>s</td>
<td>A signed short integer</td>
</tr>
<tr>
<td>S</td>
<td>An unsigned short integer</td>
</tr>
<tr>
<td>u</td>
<td>Convert to uuencode format</td>
</tr>
<tr>
<td>v</td>
<td>A short integer in VAX (little-endian) order</td>
</tr>
<tr>
<td>V</td>
<td>A long integer in VAX order</td>
</tr>
</tbody>
</table>
Perl

<table>
<thead>
<tr>
<th>x</th>
<th>A null byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Indicates &quot;go back one byte&quot;</td>
</tr>
<tr>
<td>@</td>
<td>Fill with nulls (ASCII 0)</td>
</tr>
</tbody>
</table>

**Example**

Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

<bits = pack("c", 65);
  # prints A, which is ASCII 65.
  print "bits are $bits\n";
Enter
<bits = pack("x" );
  # $bits is now a null character.
  print "bits are $bits\n";
Enter
<bits = pack("sai", 255, "T", 30 );
  # creates a seven character string on most computers'
  print "bits are $bits\n";
Enter

@array = unpack("sai", "$bits");

#Array now contains three elements: 255, A and 47.
print "Array $array[0]\n";
Enter
print "Array $array[1]\n";
Enter
print "Array $array[2]\n";
Enter
```

When above code is executed, it produces the following result:

bits are A
bits are
bits are T-
Array 255
Array T
Array 30
unshift

Description
This function places the elements from LIST, in order, at the beginning of ARRAY. This is opposite function to shift().

Syntax
Following is the simple syntax for this function:

```perl
unshift ARRAY, LIST
```

Return Value
This function returns the number of new elements in ARRAY.

Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

@array = ( 1, 2, 3, 4);

print "Value of array is @array\n" ;

unshift( @array, 20, 30, 40 );

print "Now value of array is @array\n" ;
```

When above code is executed, it produces the following result:

```
Value of array is 1 2 3 4
Now value of array is 20 30 40 1 2 3 4
```

untie

Description
This function breaks the binding between a variable and a package, undoing the association created by the tie function.
Syntax
Following is the simple syntax for this function:

untie VARIABLE

Return Value
This function returns 0 on failure and 1 on success.

Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

package MyArray;

sub TIEARRAY {
    print "TYING\n";
    bless [];
}
sub DESTROY {
    print "DESTROYING\n";
}
sub STORE{
    my ($self, $index, $value ) = @_; 
    print "STORING $value at index $index\n";
    $self[$index] = $value;
}
sub FETCH {
    my ($self, $index ) = @_; 
    print "FETCHING the value at index $index\n";
    return $self[$index];
}

package main;
```
$object = tie @x, MyArray; #@x is now a MyArray array;

print "object is a ", ref($object), "\n";

$x[0] = 'This is test'; #this will call STORE();
print $x[0], "\n"; #this will call FETCH();
print $object->FETCH(0), "\n";
untie @x #now @x is a normal array again.

When above code is executed, it produces the following result:

TYING
object is a MyArray
STORING This is test at index 0
FETCHING the value at index 0
This is test
FETCHING the value at index 0
This is test
DESTROYING

use

Description
This function imports all the functions exported by MODULE, or only those referred to by LIST, into the name space of the current package. Effectively equivalent to:

BEGIN
{
require "Module.pm";
Module->import();
}

Also used to impose compiler directives (pragmas) on the current script, although essentially these are just modules anyway.

Note that a use statement is evaluated at compile time. A require statement is evaluated at execution time.
If the VERSION argument is present between Module and LIST, then the use will call the VERSION method in class Module with the given version as an argument. The default VERSION method, inherited from the UNIVERSAL class.

**Syntax**

Following is the simple syntax for this function:

```perl
use MODULE LIST

use MODULE

use VERSION
```

**Return Value**

This function does not return any value.

**Example**

Following is the example code showing its basic usage:

```perl
use constant;
    use diagnostics;
    use integer;
    use sigtrap qw(SEGV BUS);
    use strict qw(subs vars refs);
    use subs qw(afunc blurfl);
    use warnings qw(all);
    use sort qw(stable _quicksort _mergesort);
    use v5.6.1;    # compile time version check
    use 5.6.1;     # ditto
    use 5.006_001; # ditto
```

When above code is executed, it produces the following result:
**utime**

**Description**
This function sets the access and modification times specified by ATIME and MTIME for the list of files in LIST. The values of ATIME and MTIME must be numerical. The inode modification time is set to the current time. The time must be in the numeric format (for example, seconds since January 1, 1970).

**Syntax**
Following is the simple syntax for this function:

```perl
utime ATIME, MTIME, LIST
```

**Return Value**
This function returns the number of files updated.

**Example**
Following is the example code showing its basic usage, create a file t.txt in /tmp directory and use the following program to set its modification and access time to the current time.:

```perl
#!/usr/bin/perl -w

utime(time(), time(), "/tmp/t.txt");
```

When above code is executed, it produces the following result:

---

**values**

**Description**
This function returns the list of all the values contained in HASH. In a scalar context, returns the number of values that would be returned. Uses the same iterator, and therefore order, used by the each and keys functions.

**Syntax**
Following is the simple syntax for this function:
Return Value
This function returns number of values in the hash in scalar context and list of value in list context.

Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

%hash = ('One' => 1,
         'Two' => 2,
         'Three' => 3,
         'Four' => 4);

@values = values( %hash );
print("Values are ", join("-", @values), "\n");

@keys = keys( %hash );
print("Keys are ", join("-", @keys), "\n");
```

When above code is executed, it produces the following result:

Values are 4-3-2-1
Keys are Four-Three-Two-One

vec

Description
This function uses the string specified EXPR as a vector of unsigned integers. The NUMBITS parameter is the number of bits that are reserved for each entry in the bit vector.

This must be a power of two from 1 to 32. Note that the offset is the marker for the end of the vector, and it counts back the number of bits specified to find the start. Vectors can be manipulated with the logical bitwise operators |, & and ^.
Syntax
Following is the simple syntax for this function:

vec EXPR, OFFSET, BITS

Return Value
This function returns the value of the bit field specified by OFFSET.

Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

$vec = '';
vec($vec, 3, 4) = 1;  # bits 0 to 3
vec($vec, 7, 4) = 10; # bits 4 to 7
vec($vec, 11, 4) = 3;  # bits 8 to 11
vec($vec, 15, 4) = 15; # bits 12 to 15

# As there are 4 bits per number this can
# be decoded by unpack() as a hex number
print("vec() Has a created a string of nybbles,
in hex: ", unpack("h*", $vec), "\n");
```

When above code is executed, it produces the following result:

```
vec() Has a created a string of nybbles,
in hex: 0001000a0003000f
```

wait

Description
This function Waits for a child process to terminate, returning the process ID of
the deceased process. The exit status of the process is contained in $?.

Syntax
Following is the simple syntax for this function:
Perl

wait

Return Value
This function returns -1 if there are no child processes else the Process ID of deceased process.

Example
Following is the example code showing its basic usage:

When above code is executed, it produces the following result:

waitpid

Description
This function waits for the child process with ID PID to terminate, returning the process ID of the deceased process. If PID does not exist, then it returns -1. The exit status of the process is contained in $?.

The flags can be set to various values which are equivalent to those used by the waitpid() UNIX system call. A value of 0 for FLAGS should work on all operating systems that support processes.

Syntax
Following is the simple syntax for this function:

waitpid PID, FLAGS

Return Value
This function returns -1 if process does not exist else Process ID of deceased process.

Example
Following is the example code showing its basic usage:

When above code is executed, it produces the following result:
wantarray

Description
This function returns true if the context of the currently executing function is looking for a list value. Returns false in a scalar context.

Syntax
Following is the simple syntax for this function:

```perl
wantarray
```

Return Value
This function returns undef if no context and 0 if lvalue expects a scalar.

Example
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

sub foo {
    return wantarray() ? qw(A, B, C) : '1');
}

$result = foo();  # scalar context
@result = foo();  # array context

print("foo() in a scalar context: $result\n");
print("foo() in an array context:
@result\n");
```

When above code is executed, it produces the following result:

```
foo() in a scalar context: 1
foo() in an array context:
```
**warn**

**Description**
This function prints the value of LIST to STDERR. Basically the same as the die function except that no call is made to the exit and no exception is raised within an eval statement. This can be useful to raise an error without causing the script to terminate prematurely.

If the variable $@ contains a value (from a previous eval call) and LIST is empty, then the value of $@ is printed with .\t.caught. appended to the end. If both $@ and LIST are empty, then .Warning: Something.s wrong. is printed.

**Syntax**
Following is the simple syntax for this function:

```perl
warn LIST
```

**Return Value**
This function does not return any value.

**Example**
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

warn("Unable to calculate value, using defaults instead.\n");
```

When above code is executed, it produces the following result:

```
Unable to calculate value, using defaults instead
```

**write**

**Description**
This function writes a formatted record, as specified by format to FILEHANDLE. If FILEHANDLE is omitted, then writes the output to the currently selected
default output channel. Form processing is handled automatically, adding new pages, headers, footers, and so on, as specified by the format for the filehandle.

**Syntax**
Following is the simple syntax for this function:

```plaintext
write FILEHANDLE
write
```

**Return Value**
This function returns 0 on failure and 1 on success.

**Example**
Following is the example code showing its basic usage:

When above code is executed, it produces the following result:

**-X**

**Syntax**

```
-X FILEHANDLE
-X
```

**Definition and Usage**
A file test, where X is one of the letters listed below. His unary operator takes one argument, either a filename or a filehandle, and tests the associated file to see if something is true about it.

If the argument is omitted, tests `$_`

**Return Value**
- 1 if condition is true
- 0 if condition is false

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-r</td>
<td>File is readable by effective uid/gid.</td>
</tr>
<tr>
<td>-w</td>
<td>File is writable by effective uid/gid.</td>
</tr>
<tr>
<td>-x</td>
<td>File is executable by effective uid/gid.</td>
</tr>
<tr>
<td>-o</td>
<td>File is owned by effective uid.</td>
</tr>
<tr>
<td>-R</td>
<td>File is readable by real uid/gid.</td>
</tr>
<tr>
<td>-W</td>
<td>File is writable by real uid/gid.</td>
</tr>
<tr>
<td>-X</td>
<td>File is executable by real uid/gid.</td>
</tr>
<tr>
<td>-O</td>
<td>File is owned by real uid.</td>
</tr>
<tr>
<td>-e</td>
<td>File exists.</td>
</tr>
<tr>
<td>-z</td>
<td>File has zero size (is empty).</td>
</tr>
<tr>
<td>-s</td>
<td>File has nonzero size (returns size in bytes).</td>
</tr>
<tr>
<td>-f</td>
<td>File is a plain file.</td>
</tr>
<tr>
<td>-d</td>
<td>File is a directory.</td>
</tr>
<tr>
<td>-l</td>
<td>File is a symbolic link.</td>
</tr>
<tr>
<td>-p</td>
<td>File is a named pipe (FIFO), or Filehandle is a pipe.</td>
</tr>
<tr>
<td>-S</td>
<td>File is a socket.</td>
</tr>
<tr>
<td>-b</td>
<td>File is a block special file.</td>
</tr>
<tr>
<td>-c</td>
<td>File is a character special file.</td>
</tr>
<tr>
<td>-t</td>
<td>Filehandle is opened to a tty.</td>
</tr>
<tr>
<td>-u</td>
<td>File has setuid bit set.</td>
</tr>
<tr>
<td>-g</td>
<td>File has setgid bit set.</td>
</tr>
<tr>
<td>-k</td>
<td>File has sticky bit set.</td>
</tr>
<tr>
<td>-T</td>
<td>File is an ASCII text file (heuristic guess).</td>
</tr>
<tr>
<td>-B</td>
<td>File is a &quot;binary&quot; file (opposite of -T).</td>
</tr>
<tr>
<td>-M</td>
<td>Script start time minus file modification time, in days.</td>
</tr>
<tr>
<td>-A</td>
<td>Same for access time.</td>
</tr>
</tbody>
</table>
-C Same for inode change time

**Example**
Try out following example with some file.

```perl
stat($filename);
print "Readable\n" if -r _;
print "Writable\n" if -w _;
print "Executable\n" if -x _;
print "Setuid\n" if -u _;
print "Setgid\n" if -g _;
print "Sticky\n" if -k _;
print "Text\n" if -T _;
print "Binary\n" if -B _;

# Another way of testing
if( -e $filename ){
  print " File $filename exists\n";
}
```

**y**

**Description**
This function is identical to the tr/// operator; translates all characters in
SEARCHLIST into the corresponding characters in REPLACEMENTLIST. It does
character by character conversion

**Syntax**
Following is the simple syntax for this function:

```perl
y/SEARCHLIST/REPLACEMENTLIST/
```

**Return Value**
This function returns the number of characters modified.

**Example**
Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

$string = 'the cat sat on the mat.);

# This will generate a upper case string
$string =~ y/a-z/A-Z/;

print "$string\n";
```

When above code is executed, it produces the following result:

```
THE CAT SAT ON THE MAT.
```