PERL PROGRAMMING
programming language

tutorialspoint
SIMPLY EASY LEARNING

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

Table of Contents

i

ii

iii

vi

ix

x

xi

xii

Part 1: Perl – Basics

1. INTRODUCTION

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.

- Perls 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 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**.

- 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**.

- 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.

$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:

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

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:

$perl script.pl   # Unix/Linux

or

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 admininstrator.

Al 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 designd 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 priviledge, generally a setting of 0755 works perfectly and finally you execute the above script as follows:

$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.

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:

```
# 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:

```
#!/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:

```
#!/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:

```
#!/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 docoment, just as in regular quoting. An unquoted identifier works like double quotes.

```
#!/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:

```
#!/usr/bin/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.

S.N.	Types and Description
1	**Scalar:** 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.
2	**Arrays:** Arrays are ordered lists of scalars that you access with a numeric index, which starts with 0. They are preceded by an "at" sign (@).
3	**Hashes:** Hashes are unordered sets of key/value pairs that you access using the keys as subscripts. They are preceded by a percent sign (%).

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:

Type	Value
Integer	1234
Negative integer	-100
Floating point	2000

Scientific notation	16.12E14
Hexadecimal	0xffff
Octal	0577

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:

Escape sequence	Meaning
\\	Backslash
\'	Single quote
\"	Double quote
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

\0nn	Creates Octal formatted numbers
\xnn	Creates Hexideciamal formatted numbers
\cX	Controls characters, x may be any character
\u	Forces next character to uppercase
\l	Forces next character to lowercase
\U	Forces all following characters to uppercase
\L	Forces all following characters to lowercase
\Q	Backslash all following non-alphanumeric characters
\E	End \U, \L, or \Q

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.

```
#!/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";


# Backsalash 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

5. VARIABLES

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. Finaly, 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:

```
$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.

19

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");


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 used escape sign (\) before the $ sign just to print it. Other Perl will understand it as a variable and will print its value. When exected, 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
```

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:

```
#!/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
```

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:

```
#!/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 anyother 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:

S.N.	Context and Description
1	**Scalar:** Assignment to a scalar variable evaluates the right-hand side in a scalar context.
2	**List:** Assignment to an array or a hash evaluates the right-hand side in a list context.
3	**Boolean:** Boolean context is simply any place where an expression is being evaluated to see whether it's true or false.
4	**Void:** This context not only doesn't care what the return value is, it doesn't even want a return value.

5	**Interpolative:**
	This context only happens inside quotes, or things that work like quotes.

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
$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


$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:

```
#!/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?o

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:

```perl
#!/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__

7. ARRAYS

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 exected, 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:

@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:

@days = qw/Monday

Tuesday

...

Sunday/;

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

$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:

```
#!/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:

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:

```
#!/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:

@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:

```
#!/uer/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:

Size: 51

Max Index: 50

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.

S.N.	Types and Description
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.

```
#!/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:

```
#!/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:

Thu Fri Sat

Replacing Array Elements

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

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:

```
#!/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:

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:

```
#!/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:

```
#!/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\n";

print "$string2\n";

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 SUBTROUTINE 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 variable 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 on-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:

```
#!/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:

Foods: pizza steak chicken burgers

Food at @foods[1]: pizza

Food at @foods[2]: steak

Merging Arrays

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

```
#!/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:

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

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:

%data = (-JohnPaul => 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.

$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:

```
#!/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:

```
#!/uer/bin/perl



%data = (-JohnPaul => 45, -Lisa => 30, -Kumar => 40);
```

@array = @data{-JohnPaul, -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:

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:

values %HASH

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:

```
#!/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 **if(condition)** part will be executed only when the given condition is true otherwise **else** part will be executed. So when we execute the above program, it produces the following result because here the given condition *exists($data{'Lisa'}* returns true:

Lisa is 30 years old

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:

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

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

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

@values = values %data;
$size = @values;
print "2 - Hash size:  is $size\n";
```

This will produce the following result:

1 - Hash size: is 3

2 - Hash size: is 3

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 **delete** function as shown below in the example:

```perl
#!/usr/bin/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 from of a typical decision making structure found in most of the programming languages:
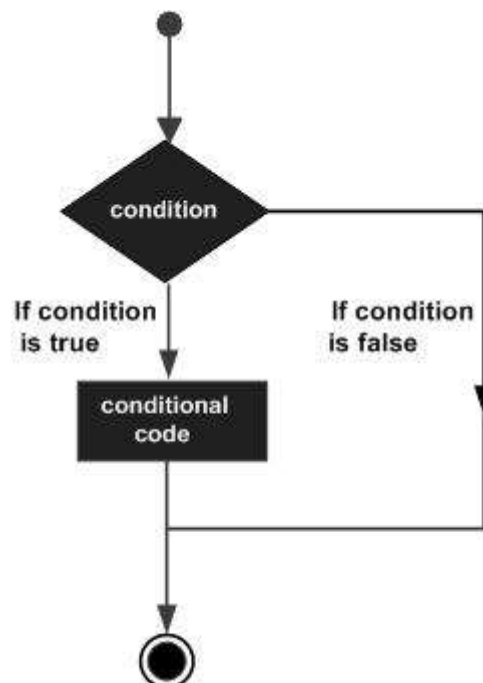
The number 0, the strings '0' and "" , the empty list () , and undef are all **false** in a boolean context and all other values are **true**. Negation of a true value by **!** or **not** returns a special false value.

Perl programming language provides the following types of conditional statements.

Statement	Description
if statement	An **if statement** consists of a boolean expression followed by one or more statements.
if...else statement	An **if statement** can be followed by an

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

```
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.

Flow Diagram
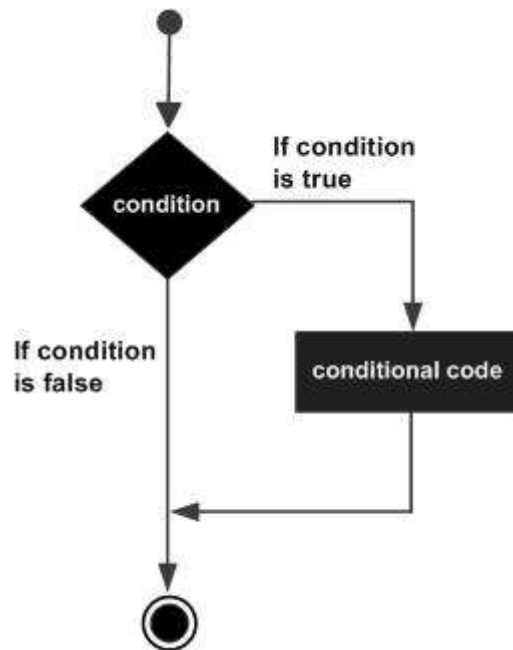
Example

```perl
#!/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:

a is less than 20

value of a is : 10

value of a is :

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:

```
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.

Flow Diagram
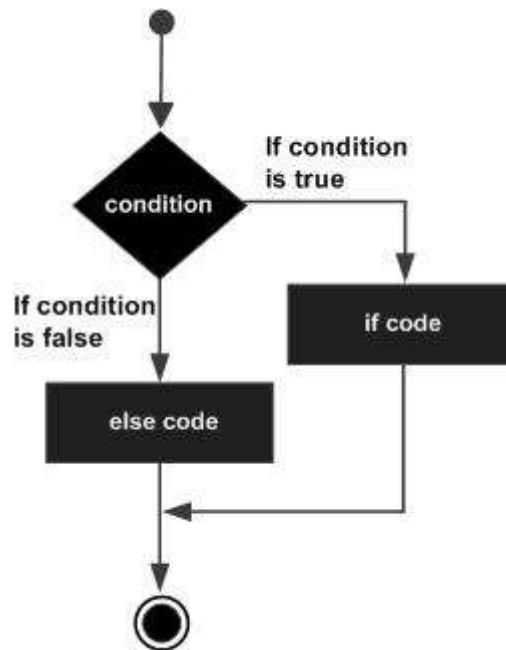
Example

```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";
```

```
}else{

   # if condition is false then print the following

    printf "a has a false value\n";

}
```

```
print "value of a is : $a\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:

```
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{

   # Executes when the 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 retruns 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
}
```

If the boolean expression evaluates to **false,** then the block of code inside the unless statement will be executed. If boolean expression evaluates to **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 **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
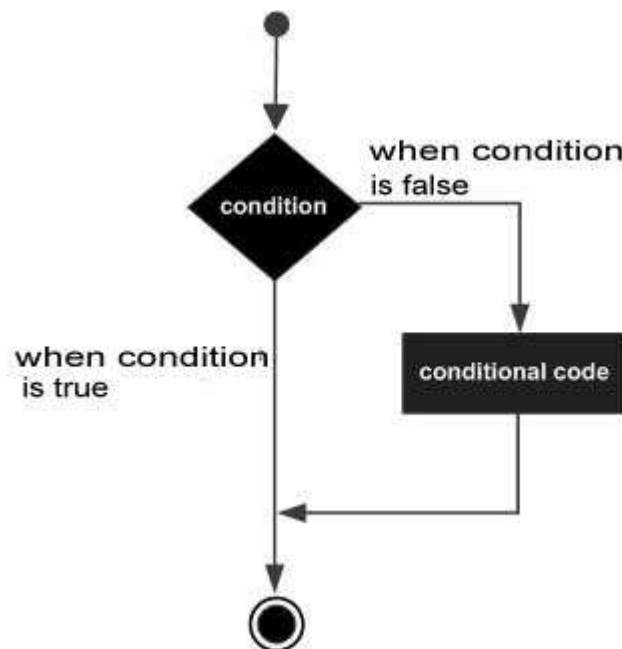
Example

```
#!/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.

Flow Diagram
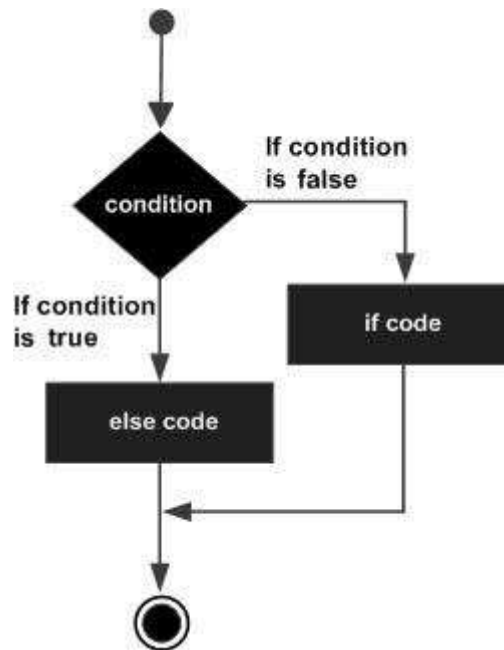
Example

```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

}
```

```
else{
```

```
   # Executes when the none of the above condition is met
```

```
}
```

Example

```
#!/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 retruns 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:

```
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 statement 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
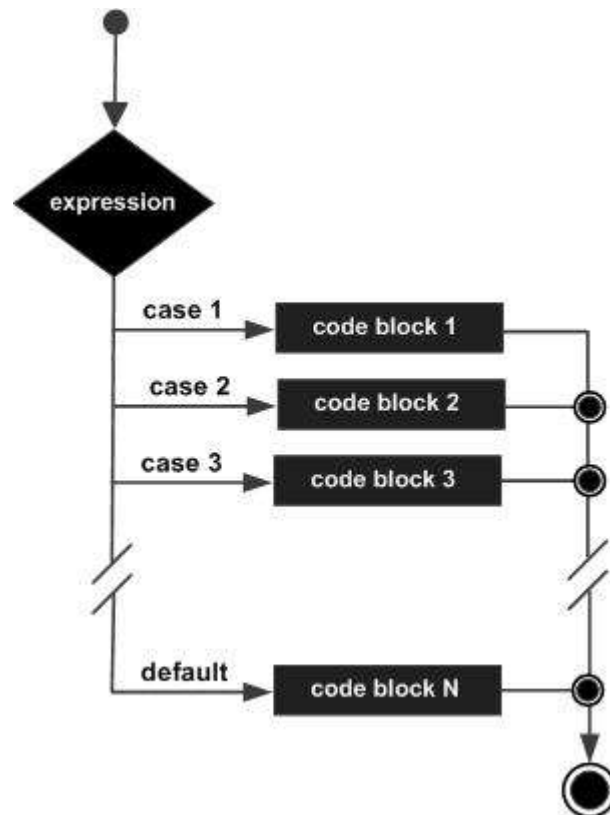
Example

```
#!/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-though 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:

```
#!/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:
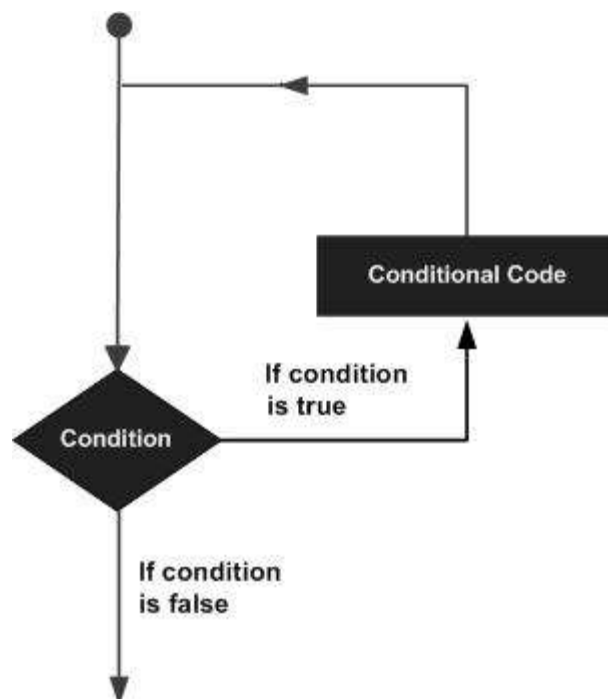
Perl programming language provides the following types of loop to handle the looping requirements.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
until loop	Repeats a statement or group of statements until a given condition becomes true. It tests the condition

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

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:

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.

Flow Diagram
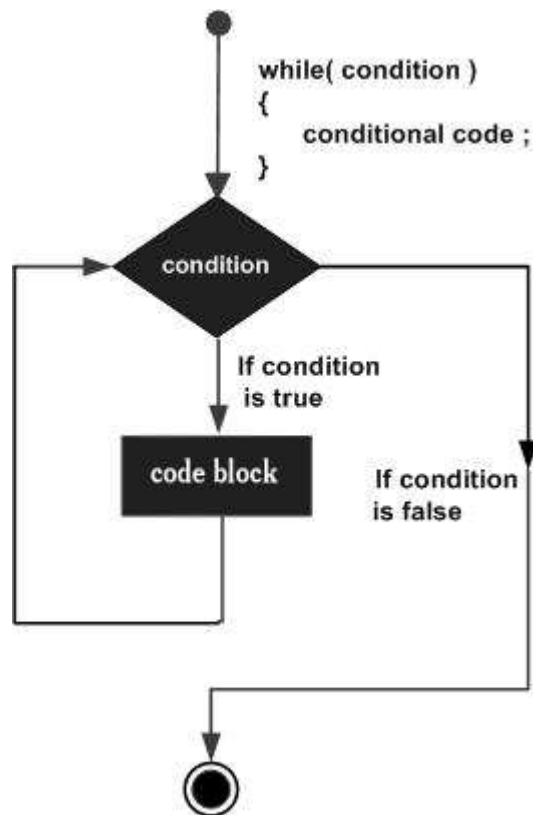
Here the key point of the *while* loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
#!/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:

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

until loop

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:

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.

Flow Diagram
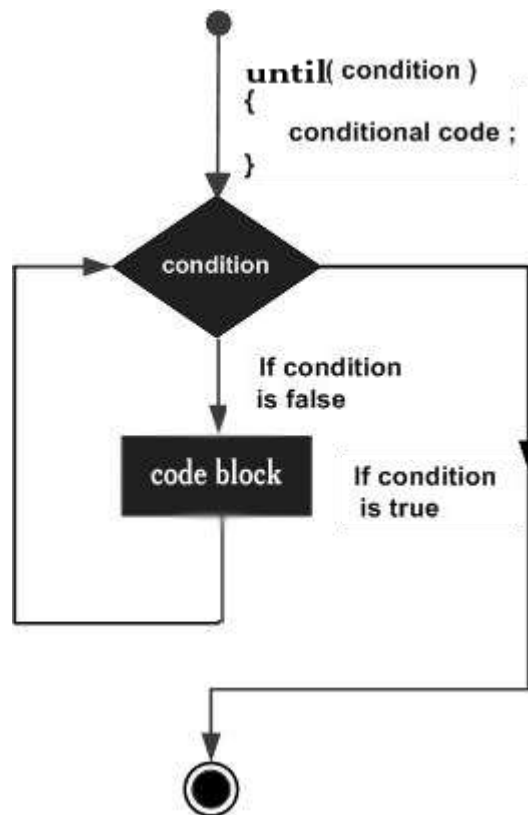
Here key point of the *until* loop is that the loop might not ever run. When the condition is tested and the result is true, the loop body will be skipped and the first statement after the until loop will be executed.

Example

```
#!/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:

Value of a: 5

Value of a: 6

Value of a: 7

Value of a: 8

Value of a: 9

Value of a: 10

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:

```
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.

Flow Diagram
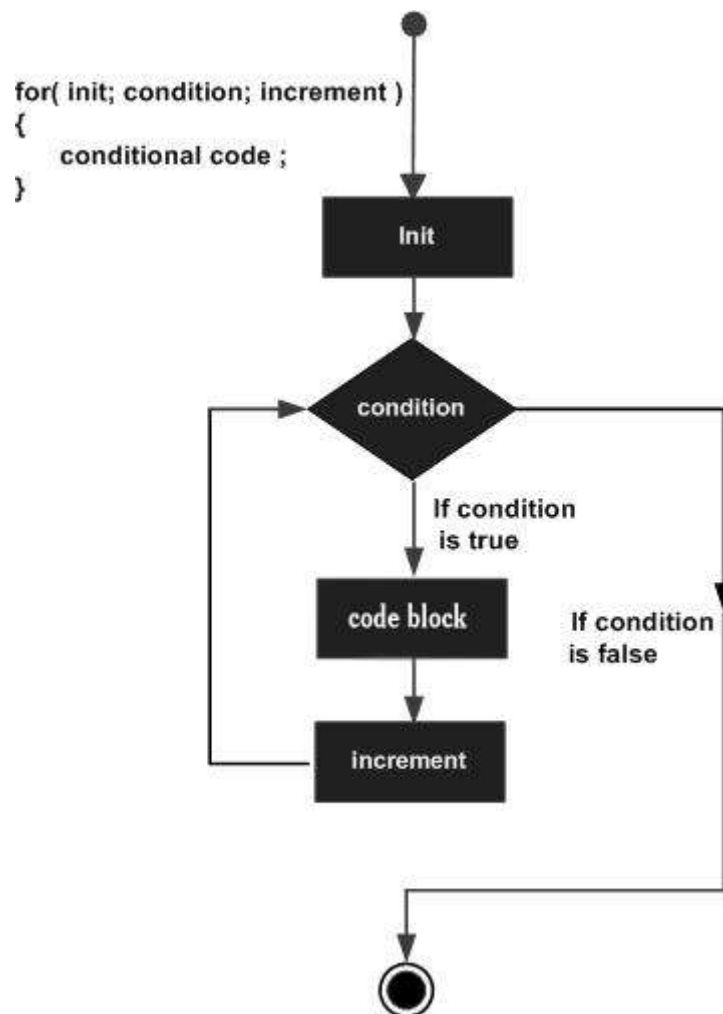
Example

```
#!/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:

foreach var (list) {

...

}

Flow Diagram
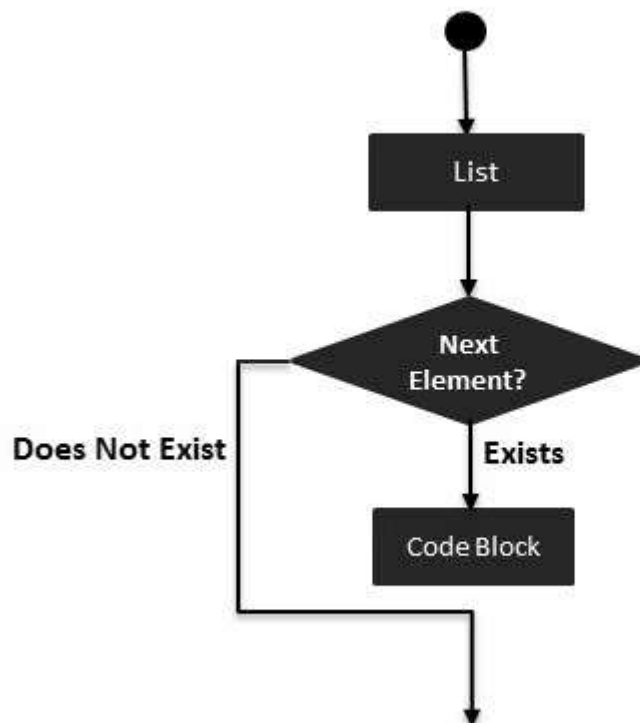
Example

```
#!/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
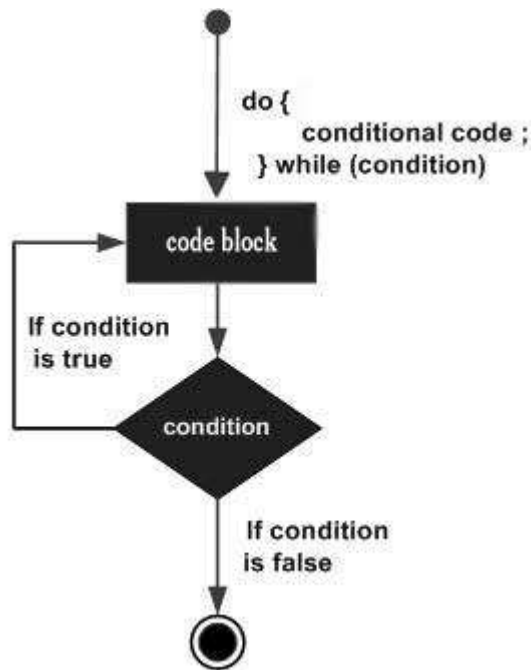
Example

```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:

```
for ( init; condition; increment ){

  for ( init; condition; increment ){

    statement(s);

  }

  statement(s);

}
```

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

```
while(condition){

  while(condition){

    statement(s);

  }

  statement(s);

}
```

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

```
do{

  statement(s);

  do{

    statement(s);

  }while( condition );


}while( condition );
```

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

```
until(condition){
```

```
  until(condition){

    statement(s);

  }

  statement(s);

}
```

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

```
foreach $a (@listA){

  foreach $b (@listB){

    statement(s);

  }

  statement(s);

}
```

Example

The following program uses a nested **while** loop to show the usage:

```
#/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.

Control Statement	Description
next statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
last statement	Terminates the loop statement and transfers execution to the statement immediately following the loop.
continue statement	A continue BLOCK, it is always executed just before the conditional is about to be evaluated again.

redo statement	The redo command restarts the loop block without evaluating the conditional again. The continue block, if any, is not executed.
goto statement	Perl supports a goto command with three forms: goto label, goto expr, and goto &name.

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
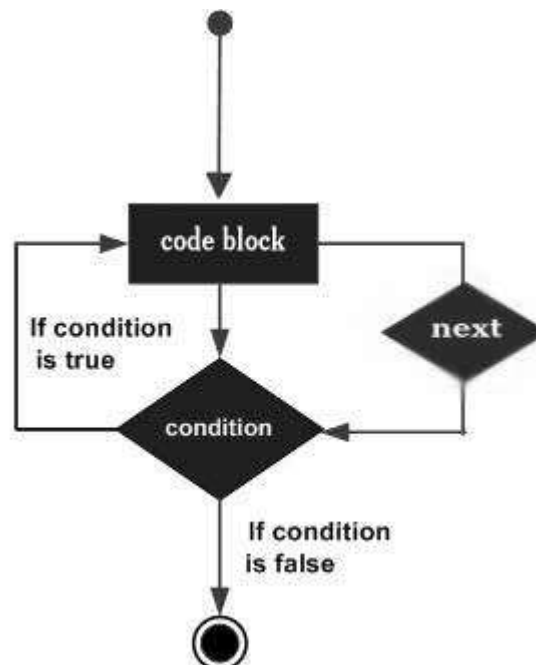
Example

```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:

```perl
#!/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";
  }
  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

Value of b : 3

Value of b : 4

value of a: 2

value of a: 3

Value of b : 1

Value of b : 2

Value of b : 3

Value of b : 4

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
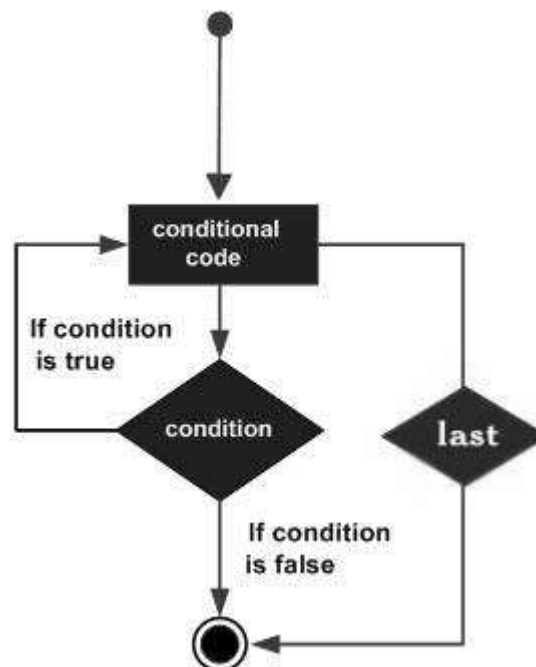
Example 1
```
#!/usr/local/bin/perl


$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

value of a: 13

value of a: 14
```

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";
```

```
   }
   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

Value of b : 3

Value of b : 4

value of a: 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:

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

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

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

Example

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

```perl
#/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:

```perl
#/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
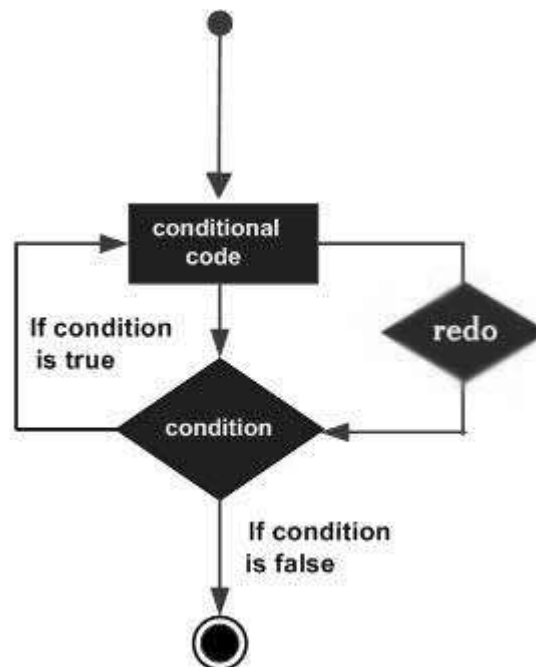
Example

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

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


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

goto statement

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

S.N.	goto type
1	**goto LABEL** The goto LABEL form jumps to the statement labeled with LABEL and resumes execution from there.
2	**goto EXPR** The goto EXPR form is just a generalization of goto LABEL. It expects the expression to return a label name and then jumps to that labeled statement.
3	**goto &NAME** It substitutes a call to the named subroutine for the currently running subroutine.

Syntax

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

goto LABEL

or

goto EXPR

or

goto &NAME

Flow Diagram
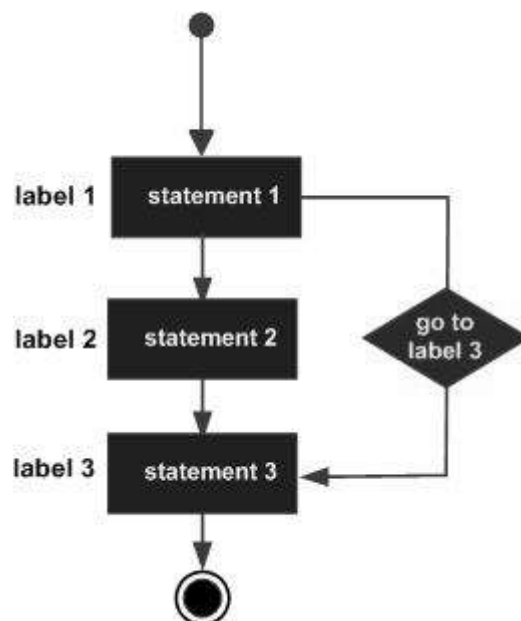
Example

The following program shows the most frequently used form of **goto** statement:

```
#/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

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
```

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:

```
#/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 = 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.

```
#!/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

Lets 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 arithmatic operators:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	$a + $b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	$a - $b will give -10
*	Multiplication - Multiplies values on either side of the operator	$a * $b will give 200
/	Division - Divides left hand operand by right hand operand	$b / $a will give 2

%	Modulus - Divides left hand operand by right hand operand and returns remainder	$b % $a will give 0
**	Exponent - Performs exponential (power) calculation on operators	$a**$b will give 10 to the power 20

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:

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

<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	($a < $b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	($a >= $b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	($a <= $b) is true.

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.

```
#!/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";
```

```
}


$c = $a <=> $b;

print "\$a <=> \$b returns $c\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";

}


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

$a > $b is true

$a >= $b is true

$a < $b is not true

$a <= $b is not true

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:

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

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";

}


$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:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	$c = $a + $b will assigned value of $a + $b into $c
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	$c += $a is equivalent to $c = $c + $a
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left	$c -= $a is equivalent to $c = $c - $a

	operand	
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	$c *= $a is equivalent to $c = $c * $a
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	$c /= $a is equivalent to $c = $c / $a
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	$c %= $a is equivalent to $c = $c % a
**=	Exponent AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand	$c **= $a is equivalent to $c = $c ** $a

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

~$a = 1100 0011

There are following Bitwise operators supported by Perl language, assume if $a = 60; and $b = 13:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	($a & $b) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in eather operand.	($a | $b) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	($a ^ $b) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the efect of 'flipping' bits.	(~$a) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	$a << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	$a >> 2 will give 15 which is 0000 1111

Example

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

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

use integer;

$a = 60;
$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:

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

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";


$c = ($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:

```
Value of $a = true and value of $b = false
Value of $a and $b = false
Value of $a && $b = false
Value of $a or $b = true
Value of $a || $b = true
Value of not($a)= 1
```

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.

Operator	Description	Example
q{ }	Encloses a string with-in single quotes	q{abcd} gives 'abcd'
qq{ }	Encloses a string with-in double quotes	qq{abcd} gives "abcd"
qx{ }	Encloses a string with-in invert quotes	qx{abcd} gives `abcd`

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:

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

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.

```
#!/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.

left   terms and list operators (leftward)

left   ->

nonassoc   ++ --

right   **

right   ! ~ \ and unary + and -

left   =~ !~

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.

```
#!/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
```

12.   DATE AND TIME

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:

```
sec,    # seconds of minutes from 0 to 61

min,    # minutes of hour from 0 to 59

hour,   # hours of day from 0 to 24

mday,   # day of month from 1 to 31

mon,    # month of year from 0 to 11

year,   # year since 1900

wday,   # days since sunday

yday,   # days since January 1st

isdst   # hours of daylight savings time
```

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

```
#!/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
```

($sec,$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.

```
#!/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:

```
#!/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 strftime()

You can use the POSIX function **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.

Specifier	Replaced by	Example
%a	Abbreviated weekday name *	Thu
%A	Full weekday name *	Thursday
%b	Abbreviated month name *	Aug
%B	Full month name *	August
%c	Date and time representation *	Thu Aug 23 14:55:02 2001
%C	Year divided by 100 and truncated to integer (`00-99`)	20
%d	Day of the month, zero-padded (`01-31`)	23
%D	Short `MM/DD/YY` date, equivalent to `%m/%d/%y`	08/23/01
%e	Day of the month, space-padded (`1-31`)	23
%F	Short `YYYY-MM-DD` date, equivalent to `%Y-%m-%d`	2001-08-23
%g	Week-based year, last two digits (`00-99`)	01
%g	Week-based year	2001
%h	Abbreviated month name * (same as `%b`)	Aug
%H	Hour in 24h format (`00-23`)	14
%I	Hour in 12h format (`01-12`)	02
%j	Day of the year (`001-366`)	235
%m	Month as a decimal number (`01-12`)	08

%M	Minute (`00-59`)	55
%n	New-line character (`'\n'`)	
%p	AM or PM designation	PM
%r	12-hour clock time *	02:55:02 pm
%R	24-hour `HH:MM` time, equivalent to `%H:%M`	14:55
%S	Second (`00-61`)	02
%t	Horizontal-tab character (`'\t'`)	
%T	ISO 8601 time format (`HH:MM:SS`), equivalent to `%H:%M:%S`	14:55
%u	ISO 8601 weekday as number with Monday as `1` (`1-7`)	4
%U	Week number with the first Sunday as the first day of week one (`00-53`)	33
%V	ISO 8601 week number (`00-53`)	34
%w	Weekday as a decimal number with Sunday as `0` (`0-6`)	4
%W	Week number with the first Monday as the first day of week one (`00-53`)	34
%x	Date representation *	08/23/01
%X	Time representation *	14:55:02
%y	Year, last two digits (`00-99`)	01

%Y	Year	2001
%z	ISO 8601 offset from UTC in timezone (1 minute=1, 1 hour=100) If timezone cannot be termined, no characters	+100
%Z	Timezone name or abbreviation * If timezone cannot be termined, no characters	CDT
%%	A % sign	%

Let's check the following example to understand the usage:

```
#!/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:

sub subroutine_name{

   body of the subroutine

}

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

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.

&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.

```
#!/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 acessed 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

tutorialspoint
SIMPLY EASY LEARNING

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) = @_;

   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";

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

Average for the given numbers : 20

Private Variables in a Subroutine

By default, all variables in Perl are global variables, which means they can be accessed from anywhere in the program. But you can create **private** variables called **lexical variables** at any time with the **my** operator.

The **my** operator confines a variable to a particular region of code in which it can be used and accessed. Outside that region, this variable cannot be used or accessed. This region is called its scope. A lexical scope is usually a block of code with a set of braces around it, such as those defining the body of the subroutine or those marking the code blocks of *if, while, for, foreach,* and *eval* statements.

Following is an example showing you how to define a single or multiple private variables using **my** operator:

```
sub somefunc {

   my $variable; # $variable is invisible outside somefunc()

   my ($another, @an_array, %a_hash); # declaring many variables at once

}
```

Let's check the following example to distinguish between global and private variables:

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

```
# 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:

Inside the function Hello, Perl!

Outside the function Hello, World!

Temporary Values via local()

The **local** is mostly used when the current value of a variable must be visible to called subroutines. A local just gives temporary values to global (meaning package) variables. This is known as *dynamic scoping*. Lexical scoping is done with my, which works more like C's auto declarations.

If more than one variable or expression is given to local, they must be placed in parentheses. This operator works by saving the current values of those variables in its argument list on a hidden stack and restoring them upon exiting the block, subroutine, or eval.

Let's check the following example to distinguish between global and local variables:

```
#!/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
```

```
   print "Value of counter is $count\n";
   $count++;
}
```

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

```
}
```

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:

```
#!/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.

```
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:

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

Now the individual variables contain the corresponding values returned by localtime() subroutine.

14.   REFERENCES

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:

$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:

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

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

$hashref = {

        'Adam' => 'Eve',

        'Clyde' => 'Bonnie',

};

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

$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:

SCALAR

ARRAY

HASH

CODE

GLOB

REF

Let's try the following example:

```
#!/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:

```
#!/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:

```perl
#!/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:

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:

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.

Using the Format

In order to invoke this format declaration, we would use the **write** keyword:

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.

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:

$~ = "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.

```
#!/usr/bin/perl


format EMPLOYEE =
===================================
@<<<<<<<<<<<<<<<<<<<<< @<<
$name $age
@#####.##
$salary
```

```
===================================
.


select(STDOUT);

$~ = EMPLOYEE;


@n = ("Ali", "Raza", "Jaffer");

@a  = (20,30, 40);

@s = (2000.00, 2500.00, 4000.000);


$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, 4000.000);

$i = 0;
```

```
foreach (@n){

  $name = $_;

  $age = $a[$i];

  $salary = $s[$i++];

  write;

}
```

Now your report will look like:

```
===================================

Name              Age

===================================

===================================

Ali               20

  2000.00

===================================

===================================

Raza              30

  2500.00

===================================

===================================

Jaffer            40

  4000.00

===================================
```

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 @<

                      $%

===================================
```

.

Now your output will look like as follows:

```
===================================
Name              Age Page 1
===================================
===================================
Ali               20
  2000.00
===================================
===================================
Raza              30
  2500.00
===================================
===================================
Jaffer            40
  4000.00
===================================
```

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:

format EMPLOYEE_BOTTOM =

End of Page @<

        $%

.

For a complete set of variables related to formating, please refer to the **Perl Special Variables** section.

16.   FILE I/O

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.

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 opend in read-only mode.

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.

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

open(DATA, "<file.txt") or die "Couldn't open file file.txt, $!";

```
while(<DATA>){
   print "$_";
}
```

Following is the syntax to open file.txt in writing mode. Here less than > sign indicates that file has to be opend 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

Entities	Definition
< or r	Read Only Access
> or w	Creates, Writes, and Truncates
>> or a	Writes, Appends, and Creates
+< or r+	Reads and Writes
+> or w+	Reads, Writes, Creates, and Truncates
+>> or a+	Reads, Writes, Appends, and Creates

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:

sysopen(DATA, "file.txt", O_RDWR);

Or to truncate the file before updating:

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.

Entities	Definition
O_RDWR	Read and Write
O_RDONLY	Read Only
O_WRONLY	Write Only
O_CREAT	Create the file
O_APPEND	Append the file
O_TRUNC	Truncate the file
O_EXCL	Stops if file already exists
O_NONBLOCK	Non-Blocking usability

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

close

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 <FILEHANDL> 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 STDIN 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.

```
#!/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.

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

unlink ("/usr/test/file1.txt");

Positioning inside a File

You can use to **tell** function to know the current position of a file and **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

tell

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:

Operator	Definition
-A	Script start time minus file last access time, in days.
-B	Is it a binary file?

-C	Script start time minus file last inode change time, in days.
-M	Script start time minus file modification time, in days.
-O	Is the file owned by the real user ID?
-R	Is the file readable by the real user ID or real group?
-S	Is the file a socket?
-T	Is it a text file?
-W	Is the file writable by the real user ID or real group?
-X	Is the file executable by the real user ID or real group?
-b	Is it a block special file?
-c	Is it a character special file?
-d	Is the file a directory?
-e	Does the file exist?
-f	Is it a plain file?
-g	Does the file have the setgid bit set?
-k	Does the file have the sticky bit set?
-l	Is the file a symbolic link?
-o	Is the file owned by the effective user ID?
-p	Is the file a named pipe?
-r	Is the file readable by the effective user or group ID?

-s	Returns the size of the file, zero size = empty file.
-t	Is the filehandle opened by a TTY (terminal)?
-u	Does the file have the setuid bit set?
-w	Is the file writable by the effective user or group ID?
-x	Is the file executable by the effective user or group ID?
-z	Is the file size zero?

Following are the standard functions used to play with directories.

```
opendir DIRHANDLE, EXPR  # To open a directory

readdir DIRHANDLE      # To read a directory

rewinddir DIRHANDLE     # Positioning pointer to the begining

telldir DIRHANDLE      # Returns current position of the dir

seekdir DIRHANDLE, POS  # Pointing pointer to POS inside dir

closedir DIRHANDLE      # Closing a directory.
```

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.

```
#!/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.

```
#!/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.

```
#!/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:

```
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:

```
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:

```
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:

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 **?:**

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:

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:

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.

```
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:

```
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.

```
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.

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:

```
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

dickory

doc

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.

$_	The default input and pattern-searching space.
$ARG	

$. $NR	The current input line number of the last filehandle that was read. An explicit close on the filehandle resets the line number.
$/ $RS	The input record separator; newline by default. If set to the null string, it treats blank lines as delimiters.
$, $OFS	The output field separator for the print operator.
$\ $ORS	The output record separator for the print operator.
$" $LIST_SEPARATOR	Like "$," except that it applies to list values interpolated into a double-quoted string (or similar interpreted string). Default is a space.
$; $SUBSCRIPT_SEPARATOR	The subscript separator for multidimensional array emulation. Default is "\034".
$^L $FORMAT_FORMFEED	What a format outputs to perform a formfeed. Default is "\f".
$: $FORMAT_LINE_BREAK_CHARACTERS	The current set of characters after which a string may be broken to fill continuation fields (starting with ^) in a format. Default is "\n"".
$^A $ACCUMULATOR	The current value of the write accumulator for format lines.

$#	Contains the output format for printed numbers (deprecated).
$OFMT	
$?	The status returned by the last pipe close, backtick (``) command, or system operator.
$CHILD_ERROR	
$!	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.
$OS_ERROR or $ERRNO	
$@	The Perl syntax error message from the last eval command.
$EVAL_ERROR	
$$	The pid of the Perl process running this script.
$PROCESS_ID or $PID	
$<	The real user ID (uid) of this process.
$REAL_USER_ID or $UID	
$>	The effective user ID of this process.
$EFFECTIVE_USER_ID or $EUID	
$(	The real group ID (gid) of this process.
$REAL_GROUP_ID or $GID	
$)	The effective gid of this process.
$EFFECTIVE_GROUP_ID or $EGID	

$0 $PROGRAM_NAME	Contains the name of the file containing the Perl script being executed.
$[	The index of the first element in an array and of the first character in a substring. Default is 0.
$] $PERL_VERSION	Returns the version plus patchlevel divided by 1000.
$^D $DEBUGGING	The current value of the debugging flags.
$^E $EXTENDED_OS_ERROR	Extended error message on some platforms.
$^F $SYSTEM_FD_MAX	The maximum system file descriptor, ordinarily 2.
$^H	Contains internal compiler hints enabled by certain pragmatic modules.
$^I $INPLACE_EDIT	The current value of the inplace-edit extension. Use undef to disable inplace editing.
$^M	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 more information.

$^O $OSNAME	Contains the name of the operating system that the current Perl binary was compiled for.
$^P $PERLDB	The internal flag that the debugger clears so that it doesn't debug itself.
$^T $BASETIME	The time at which the script began running, in seconds since the epoch.
$^W $WARNING	The current value of the warning switch, either true or false.
$^X $EXECUTABLE_NAME	The name that the Perl binary itself was executed as.
$ARGV	Contains the name of the current file when reading from .

Global Array Special Variables

@ARGV	The array containing the command-line arguments intended for the script.
@INC	The array containing the list of places to look for Perl scripts to be evaluated by the do, require, or use constructs.
@F	The array into which the input lines are split when the -a command-line switch is given.

Global Hash Special Variables

%INC	The hash containing entries for the filename of each file that has been included via do or require.

%ENV	The hash containing your current environment.
%SIG	The hash used to set signal handlers for various signals.

Global Special Filehandles

ARGV	The special filehandle that iterates over command line filenames in @ARGV. Usually written as the null filehandle in <>.
STDERR	The special filehandle for standard error in any package.
STDIN	The special filehandle for standard input in any package.
STDOUT	The special filehandle for standard output in any package.
DATA	The special filehandle that refers to anything following the __END__ token in the file containing the script. Or, the special filehandle for anything following the __DATA__ token in a required file, as long as you're reading data in the same package __DATA__ was found in.
_ (underscore)	The special filehandle used to cache the information from the last stat, lstat, or file test operator.

Global Special Constants

__END__	Indicates the logical end of your program. Any following text is ignored, but may be read via the DATA filehandle.
__FILE__	Represents the filename at the point in your program where it's used. Not interpolated into strings.
__LINE__	Represents the current line number. Not interpolated into strings.

__PACKAGE__	Represents the current package name at compile time, or undefined if there is no current package. Not interpolated into strings.

Regular Expression Special Variables

$digit	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.
$& $MATCH	The string matched by the last successful pattern match.
$` $PREMATCH	The string preceding whatever was matched by the last successful pattern match.
$' $POSTMATCH	The string following whatever was matched by the last successful pattern match.
$+ $LAST_PAREN_MATCH	The last bracket matched by the last search pattern. This is useful if you don't know which of a set of alternative patterns was matched. For example: /Version: (.*)|Revision: (.*)/ && ($rev = $+);

Filehandle Special Variables

$| $OUTPUT_AUTOFLUSH	If set to nonzero, forces an fflush(3) after every write or print on the currently selected output channel.
$% $FORMAT_PAGE_NUMBER	The current page number of the currently selected output channel.
$=	The current page length (printable lines) of the

$FORMAT_LINES_PER_PAGE	currently selected output channel. Default is 60.
$- $FORMAT_LINES_LEFT	The number of lines left on the page of the currently selected output channel.
$~ $FORMAT_NAME	The name of the current report format for the currently selected output channel. Default is the name of the filehandle.
$^ $FORMAT_TOP_NAME	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.

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 curlies.

- 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:

open(FOO,$foo) || die "Can't open $foo: $!";

Is better than -

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,

print "Starting analysis\n" if $verbose;

Is better than -

$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:

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 regexp 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:

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.

21. REGULAR EXPRESSIONS

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:

```
#!/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:

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

Modifier	Description
i	Makes the match case insensitive.
m	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.
o	Evaluates the expression only once.
s	Allows use of . to match a newline character.
x	Allows you to use white space in the expression for clarity.
g	Globally finds all matches.
cg	Allows the search to continue even after a global match fails.

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 footbrdige/;


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: footbrdige

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.

Modifier	Description
i	Makes the match case insensitive.
m	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.
o	Evaluates the expression only once.
s	Allows use of . to match a newline character.
x	Allows you to use white space in the expression for clarity.
g	Replaces all occurrences of the found expression with the replacement text.
e	Evaluates the replacement as if it were a Perl statement, and uses its return value as the replacement text.

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 **uc** function.

```
$string =~ tr/a-z/A-Z/;
```

Translation Operator Modifiers

Following is the list of operators related to translation.

Modifier	Description
c	Complements SEARCHLIST.
d	Deletes found but unreplaced characters.
s	Squashes duplicate replaced characters.

The /d modifier deletes the characters matching SEARCHLIST that do not have a corresponding entry in REPLACEMENTLIST. For example:

```
#!/usr/bin/perl


$string = 'the cat sat on the mat.';

$string =~ tr/a-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:

```
#!/usr/bin/perl

$string = 'food';
$string = 'food';
$string =~ tr/a-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.

Pattern	Description
^	Matches beginning of line.
$	Matches end of line.
.	Matches any single character except newline. Using m option allows it to match newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets.
*	Matches 0 or more occurrences of preceding expression.

+	Matches 1 or more occurrence of preceding expression.
?	Matches 0 or 1 occurrence of preceding expression.
{ n}	Matches exactly n number of occurrences of preceding expression.
{ n,}	Matches n or more occurrences of preceding expression.
{ n, m}	Matches at least n and at most m occurrences of preceding expression.
a| b	Matches either a or b.
\w	Matches word characters.
\W	Matches nonword characters.
\s	Matches whitespace. Equivalent to [\t\n\r\f].
\S	Matches nonwhitespace.
\d	Matches digits. Equivalent to [0-9].
\D	Matches nondigits.
\A	Matches beginning of string.
\Z	Matches end of string. If a newline exists, it matches just before newline.
\z	Matches end of string.
\G	Matches point where last match finished.
\b	Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.

\B	Matches nonword boundaries.
\n, \t, etc.	Matches newlines, carriage returns, tabs, etc.
\1...\9	Matches nth grouped subexpression.
\10	Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.
[aeiou]	Matches a single character in the given set
[^aeiou]	Matches a single character outside the given set

The ^ metacharacter matches the beginning of the string and the $ metasymbol matches the end of the string. Here are some brief examples.

```
# 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+$/
```

Lets 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:

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:

if (($string =~ /Martin Brown/) || ($string =~ /Sharon Brown/))

This could be written as follows

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.

$string =~ /(\S+)\s+(\S+)/;

and

$string =~ /\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.

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:

#!/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

Example	Description
Perl	Match "Perl".

Character Classes

Example	Description

[Pp]ython	Matches "Python" or "python"
rub[ye]	Matches "ruby" or "rube"
[aeiou]	Matches any one lowercase vowel
[0-9]	Matches any digit; same as [0123456789]
[a-z]	Matches any lowercase ASCII letter
[A-Z]	Matches any uppercase ASCII letter
[a-zA-Z0-9]	Matches any of the above
[^aeiou]	Matches anything other than a lowercase vowel
[^0-9]	Matches anything other than a digit

Special Character Classes

Example	Description
.	Matches any character except newline
\d	Matches a digit: [0-9]
\D	Matches a nondigit: [^0-9]
\s	Matches a whitespace character: [\t\r\n\f]
\S	Matches nonwhitespace: [^ \t\r\n\f]
\w	Matches a single word character: [A-Za-z0-9_]
\W	Matches a nonword character: [^A-Za-z0-9_]

Repetition Cases

Example	Description
ruby?	Matches "rub" or "ruby": the y is optional
ruby*	Matches "rub" plus 0 or more ys
ruby+	Matches "rub" plus 1 or more ys
\d{3}	Matches exactly 3 digits
\d{3,}	Matches 3 or more digits
\d{3,5}	Matches 3, 4, or 5 digits

Nongreedy Repetition

This matches the smallest number of repetitions:

Example	Description
<.*>	Greedy repetition: matches "<python>perl>"
<.*?>	Nongreedy: matches "<python>" in "<python>perl>"

Grouping with Parentheses

Example	Description
\D\d+	No group: + repeats \d
(\D\d)+	Grouped: + repeats \D\d pair
([Pp]ython(,)?)+	Match "Python", "Python, python, python", etc.

Backreferences

This matches a previously matched group again:

Example	Description

([Pp])ython&\1ails	Matches python&pails or Python&Pails
(['"])[^\1]*\1	Single or double-quoted string. \1 matches whatever the 1st group matched. \2 matches whatever the 2nd group matched, etc.

Alternatives

Example	Description
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.

Example	Description
^Python	Matches "Python" at the start of a string or internal line
Python$	Matches "Python" at the end of a string or line
\APython	Matches "Python" at the start of a string
Python\Z	Matches "Python" at the end of a string
\bPython\b	Matches "Python" at a word boundary
\brub\B	\B is nonword boundary: match "rub" in "rube" and "ruby" but not alone
Python(?=!)	Matches "Python", if followed by an exclamation point
Python(?!!)	Matches "Python", if not followed by an exclamation point

Special Syntax with Parentheses

Example	Description
R(?#comment)	Matches "R". All the rest is a comment
R(?i)uby	Case-insensitive while matching "uby"
R(?i:uby)	Same as above
rub(?:y|le))	Group only without creating \1 backreference

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.

```
#!/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\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\n** 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** 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

$make install

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:

```
#!/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\n** in the header part of the email. Following is the script, which will take care of sending HTML formatted email:

```
#!/usr/bin/perl
use MIME::Lite;
```

```perl
$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:

```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,
               Type     => 'multipart/mixed'
               );
```

```
# Add your text message.
$msg->attach(Type        => 'text',
        Data      => $message
        );
```

```
# Specify your file as attachement.
$msg->attach(Type        => 'image/gif',
        Path      => '/tmp/logo.gif',
        Filename  => 'logo.gif',
        Disposition => 'attachment'
        );
$msg->send;
```

print "Email Sent Successfully\n";

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:

$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

23. SOCKET PROGRAMMING

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 the complete sequence of the calls used by Client and Server to communicate with each other:
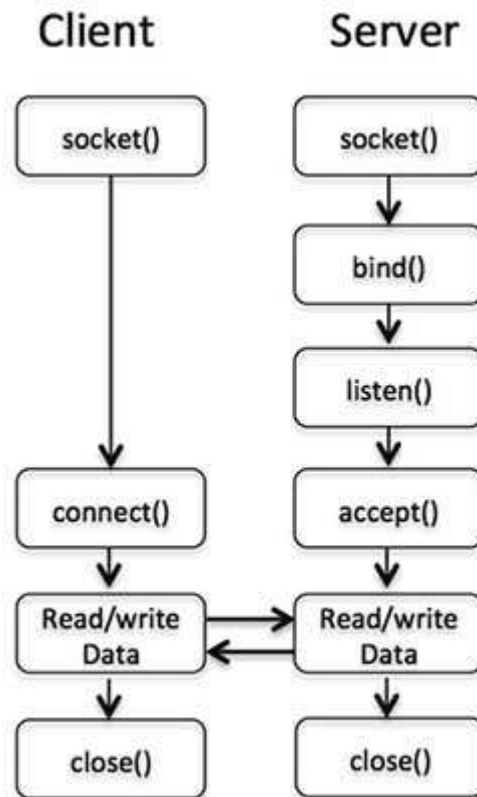
Server Side Socket Calls

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:

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.

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:

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:

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:

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 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.

while(1) {

  accept(NEW_SOCKET, SOCKT);

  ....

}

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:

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.

$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 = 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 $!\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 recieved 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,(getprotobyname('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:

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:

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.

$object = new Person("Mohammad", "Saleem", 23234345);

You can use simple hash in your consturctor if you don't want to assign any value to any class variable. For example:

package Person;

sub new

{

```perl
   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,
```

```perl
        _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;
}
sub setFirstName {
    my ( $self, $firstName ) = @_;
    $self->{_firstName} = $firstName if defined($firstName);
    return $self->{_firstName};
}


sub getFirstName {
    my( $self ) = @_;
    return $self->{_firstName};
}
1;
```

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};

}


# Add more methods

sub setLastName{

   my ( $self, $lastName ) = @_;

   $self->{_lastName} = $lastName if defined($lastName);

   return $self->{_lastName};

}


sub getLastName {

   my( $self ) = @_;

   return $self->{_lastName};

}


1;
```

Now let's again try to use Employee object in our main.pl file and execute 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

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:

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 implemnetation of Inheritance.

package MySubClass;


@ISA = qw( MyClass );
```

```perl
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->SUPER::MyMethod();
  print "   MySubClass::MyMethod called!\n";
}

# Here is the main program using above classes.
package main;

print "Invoke MyClass method\n";

$myObject = MyClass->new();
$myObject->MyMethod();

print "Invoke MySubClass method\n";
```

```
$myObject2 = MySubClass->new();

$myObject2->MyMethod();


print "Create a scoped object\n";

{

        my $myObject2 = MyClass->new();

}
# Destructor is called automatically here


print "Create and undef an object\n";

$myObject3 = MyClass->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

25. DATABASE ACCESS

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 architure diagram.
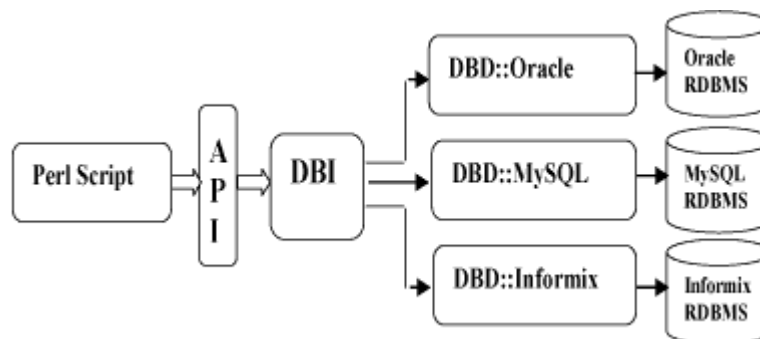
Here DBI is responsible of taking all SQL commands through the API, (i.e., Application Programming Interface) and to dispatch them to the appropriate driver for actual execution. And finally, DBI is responsible of taking results from the driver and giving back it to the calling scritp.

Notation and Conventions

Throughout this chapter following notations will be used and it is recommended that you should also follow the same convention.

$dsn   Database source name

$dbh   Database handle object

$sth   Statement handle object

$h   Any of the handle types above ($dbh, $sth, or $drh)

$rc   General Return Code (boolean: true=ok, false=error)

212

$rv   General Return Value (typically an integer)

@ary   List of values returned from the database.

$rows   Number of rows processed (if available, else -1)

$fh   A filehandle

undef   NULL values are represented by undefined values in Perl

\%attr Reference to a hash of attribute values passed to methods

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":

```
#!/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:

- Prearing 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 Stattement 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.

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:

my $first_name = "john";

my $last_name = "poul";

my $sex = "M";

```perl
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 databasse 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:

- Prearing 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 Stattement handle. This will be done using **finish()** API.

```perl
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:

- Prearing 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 Stattement 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:

$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.

$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.

- Prearing 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 Stattement handle. This will be done using **finish()** API.

- If everything goes fine then **commit** this operation otherwise you can **rollback** complete transaction.

$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.

$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 orignal position.

Here is a simple example to call **commit** API.

$dbh->commit or die $dbh->errstr;

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.

$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.

$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:

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 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:

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:

$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:

```
$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:

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:

$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

@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

%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

@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

$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.

$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:

$quoted = $dbh->quote("one\ntwo\0three")

may produce results which will be equivalent to

CONCAT('one', CHAR(12), 'two', CHAR(0), 'three')

Methods Common to All Handles

err

$rv = $h->err;

or

$rv = $DBI::err

or

$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

$str = $h->errstr;

or

$str = $DBI::errstr

or

$str = $h->errstr

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

$rv = $h->rows;

or

$rv = $DBI::rows

This returns the number of rows effected by previous SQL statement and equivalent to $DBI::rows.

trace

$h->trace($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:

```
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.

26. CGI PROGRAMMING

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 is 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, lets see what happens when we click a hyper link 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.

CGI Architecture Diagram

HTTP Protocol
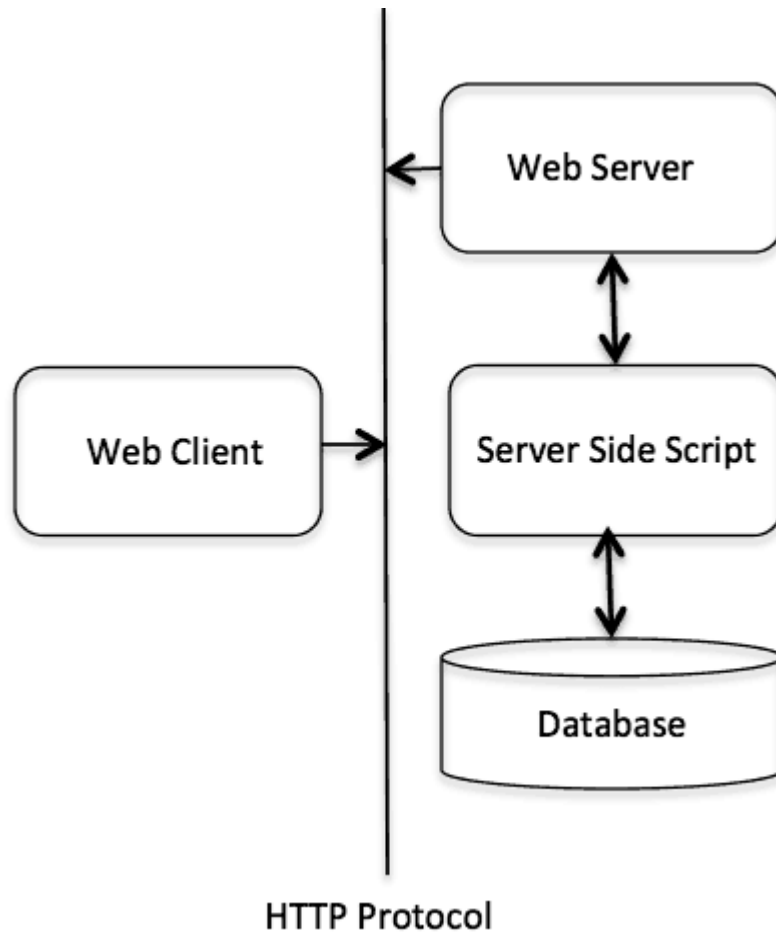
Web Server Support and Configuration

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 extention 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 undertood basic concept of CGI and you can write many complicated CGI programs using Perl. This script can interact with any other exertnal 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.

Header	Description

Content-type: String	A MIME string defining the format of the content being returned. Example is Content-type:text/html
Expires: Date String	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.
Location: URL String	The URL that should be returned instead of the URL requested. You can use this filed to redirect a request to any other location.
Last-modified: String	The date of last modification of the file.
Content-length: String	The length, in bytes, of the data being returned. The browser uses this value to report the estimated download time for a file.
Set-Cookie: String	Set the cookie passed through the *string*

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.

Variable Names	Description
CONTENT_TYPE	The data type of the content. Used when the client is sending attached content to the server. For example file upload, etc.
CONTENT_LENGTH	The length of the query information. It's available only for POST requests
HTTP_COOKIE	Returns the set cookies in the form of key & value pair.
HTTP_USER_AGENT	The User-Agent request-header field contains information about the user agent originating the request. Its name of the web browser.

PATH_INFO	The path for the CGI script.
QUERY_STRING	The URL-encoded information that is sent with GET method request.
REMOTE_ADDR	The IP address of the remote host making the request. This can be useful for logging or for authentication purpose.
REMOTE_HOST	The fully qualified name of the host making the request. If this information is not available then REMOTE_ADDR can be used to get IR address.
REQUEST_METHOD	The method used to make the request. The most common methods are GET and POST.
SCRIPT_FILENAME	The full path to the CGI script.
SCRIPT_NAME	The name of the CGI script.
SERVER_NAME	The server's hostname or IP Address.
SERVER_SOFTWARE	The name and version of the software the server is running.

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)

{

 print "<b>$_</b>: $ENV{$_}<br>\n";

}


1;
```

tutorialspoint
SIMPLYEASYLEARNING

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 achived through HTTP header.

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\"\r\n";

print "Content-Disposition: attachment; filename=\"FileName\"\r\n\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 defualt 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 alongwith 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.

```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 "GET")
{
  $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;
```

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.

```
<FORM action="/cgi-bin/hello_get.cgi" method="GET">

First Name: <input type="text" name="first_name">  <br>


Last Name: <input type="text" name="last_name">

<input type="submit" value="Submit">

</FORM>
```

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.

First Name:

Last Name: [Submit]

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.

```
<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 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: [   ] [Submit]

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.

```
<form action="/cgi-bin/checkbox.cgi" method="POST" target="_blank">

<input type="checkbox" name="maths" value="on"> Maths

<input type="checkbox" name="physics" value="on"> Physics

<input type="submit" value="Select Subject">
```

</form>

The result of this code is the following form:

☐ Maths ☐ Physics [Submit]

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";
}
```

```perl
if( $FORM{physics} ){
   $physics_flag ="ON";
}else{
   $physics_flag ="OFF";
}


print "Content-type:text/html\r\n\r\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.

```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;
}
$subject = $FORM{subject};

print "Content-type:text/html\r\n\r\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>";
```

```perl
print "</html>";
```

```perl
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">
```

```html
<textarea name="textcontent" cols=40 rows=4>
```

Type your text here...

```html
</textarea>
```

```html
<input type="submit" value="Submit">
```

```html
</form>
```

The result of this code is the following form:

Below is the **textarea.cgi** script to handle input given by the web browser.

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

```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);
```

```perl
foreach $pair (@pairs)
{
  ($name, $value) = split(/=/, $pair);
  $value =~ tr/+/ /;
  $value =~ s/%(..)/pack("C", hex($1))/eg;
  $FORM{$name} = $value;
}
$text_content = $FORM{textcontent};


print "Content-type:text/html\r\n\r\n";
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

```html
<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:

Below is the **dropdown.cgi** script to handle input given by 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;
   $FORM{$name} = $value;
}
$subject = $FORM{dropdown};

print "Content-type:text/html\r\n\r\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 retrviewed 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:

```
#!/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";

print "Set-Cookie:Domain=www.tutorialspoint.com;\n";

print "Set-Cookie:Path=/perl;\n";

print "Content-type:text/html\r\n\r\n";

..........Rest of the HTML Content goes here....

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.

```
#!/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 once.

- **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 modules'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"
}
```

```
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:

```
#!/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.

```
#!/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.

```
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:

```
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 descritpion 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:

```
#!/usr/bin/perl


@files = `ls -l`;


foreach $file (@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

-rw-r--r-- 1 root root 574 Sep 17 15:16 index.htm

250

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

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 >:

```
#!/usr/bin/perl


system( "ls -l")


1;
```

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:

```
#!/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.


1;
```

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/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 create 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";
```

```
}
```

```
1;
```

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";
```

```
}
```

```
1;
```

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:

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

```
kill('INT', 104, 102);
```

```
1;
```

29. EMBEDDED DOCUMENTATION

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:

```
#!/usr/bin/perl


print "Hello, World\n";


=head1 Hello, World Example
```

This example demonstrate very basic syntax of Perl.

```
=cut


print "Hello, Universe\n";
```

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.

```
#!/usr/bin/perl


print "Hello, World\n";


while(<DATA>){

  print $_;
```

```
}
```

`__END__`

=head1 Hello, World Example

This example demonstrate very basic syntax of Perl.

print "Hello, Universe\n";

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

Hello, World

=head1 Hello, World Example

This example demonstrate very basic syntax of Perl.

print "Hello, Universe\n";

Let's take one more example for the same code without reading DATA part:

`#!/usr/bin/perl`

print "Hello, World\n";

`__END__`

=head1 Hello, World Example

This example demonstrate 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:

=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:

=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:

=head2 An Example List

=over 4

=item * This is a bulleted list.

=item * Here's another item.

=back

=begin html

<p>

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.

</p>

=end html

When you convert the above POD into HTML using pod2html, it will produce the following result:

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.

30.　FUNCTIONS REFERENCES

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 interger 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

abs

Return Value

This function returns the absolute value of its argument.

Example
Following is the example code showing its basic usage:

```
#!/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" ;
```

259

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

Abs value of par1 is 10.25

Abs value of par2 is 20

Abs value of par3 is 0

accept

Description

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.

Syntax

Following is the simple syntax for this function:

accept NEWSOCKET,GENERICSOCKET

Return Value

This function returns 0 on failure and Packed address of remote host on success.

Example

Following is the example code showing script to create a server:

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

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

alarm

Description

This function sets the "alarm," causing the current process to receive a SIGALRM signal in EXPR seconds. If EXPR is omitted, the value of $_ is used instead.

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:

```
#!/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 -PI to PI.

Syntax

Following is the simple syntax for this function:

atan2 Y,X

Return Value

This function returns arctangent of Y/X in the range -PI to PI.

Example

Following is the example code showing its basic usage:

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


$Y = 30;

$X = 60;


$ret_val = atan2 ($Y, $X );


print "atan2 of 30/60 is : $retval\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:

```perl
#!/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:

```
#!/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:

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.:

```
#!/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

caller

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:

```
#!/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

chdir

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 funcation 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:

#!/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:

```perl
#!/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\n";
print " Number of characters removed : $retval\n";
```

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 );

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

funcation 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

chr

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

DEC	OCT	HEX	BIN	Symbol	HTML Code	Description
0	000	00	00000000	NUL	&#000;	Null char
1	001	01	00000001	SOH	&#001;	Start of Heading
2	002	02	00000010	STX	&#002;	Start of Text
3	003	03	00000011	ETX	&#003;	End of Text
4	004	04	00000100	EOT	&#004;	End of Transmission
5	005	05	00000101	ENQ	&#005;	Enquiry
6	006	06	00000110	ACK	&#006;	Acknowledgment
7	007	07	00000111	BEL	&#007;	Bell

8	010	08	00001000	BS	&#008;	Back Space
9	011	09	00001001	HT			Horizontal Tab
10	012	0A	00001010	LF	
	Line Feed
11	013	0B	00001011	VT	&#011;	Vertical Tab
12	014	0C	00001100	FF		Form Feed
13	015	0D	00001101	CR		Carriage Return
14	016	0E	00001110	SO	&#014;	Shift Out / X-On
15	017	0F	00001111	SI	&#015;	Shift In / X-Off
16	020	10	00010000	DLE	&#016;	Data Line Escape
17	021	11	00010001	DC1	&#017;	Device Control 1 (oft. XON)
18	022	12	00010010	DC2	&#018;	Device Control 2
19	023	13	00010011	DC3	&#019;	Device Control 3 (oft. XOFF)
20	024	14	00010100	DC4	&#020;	Device Control 4
21	025	15	00010101	NAK	&#021;	Negative Acknowledgement
22	026	16	00010110	SYN	&#022;	Synchronous Idle
23	027	17	00010111	ETB	&#023;	End of Transmit Block
24	030	18	00011000	CAN	&#024;	Cancel
25	031	19	00011001	EM	&#025;	End of Medium
26	032	1A	00011010	SUB	&#026;	Substitute
27	033	1B	00011011	ESC	&#027;	Escape
28	034	1C	00011100	FS	&#028;	File Separator

29	035	1D	00011101	GS	&#029;	Group Separator
30	036	1E	00011110	RS	&#030;	Record Separator
31	037	1F	00011111	US	&#031;	Unit Separator
32	040	20	00100000		 	Space
33	041	21	00100001	!	!	Exclamation mark
34	042	22	00100010	"	"	Double quotes
35	043	23	00100011	#	#	Number
36	044	24	00100100	$	$	Dollar
37	045	25	00100101	%	%	Procenttecken
38	046	26	00100110	&	&	Ampersand
39	047	27	00100111	'	'	Single quote
40	050	28	00101000	(	(	Open parenthesis
41	051	29	00101001	)	)	Close parenthesis
42	052	2A	00101010	*	*	Asterisk
43	053	2B	00101011	+	+	Plus
44	054	2C	00101100	,	,	Comma
45	055	2D	00101101	-	-	Hyphen
46	056	2E	00101110	.	.	Period, dot or full stop
47	057	2F	00101111	/	/	Slash or divide
48	060	30	00110000	0	0	Zero
49	061	31	00110001	1	1	One

50	062	32	00110010	2	2	Two
51	063	33	00110011	3	3	Three
52	064	34	00110100	4	4	Four
53	065	35	00110101	5	5	Five
54	066	36	00110110	6	6	Six
55	067	37	00110111	7	7	Seven
56	070	38	00111000	8	8	Eight
57	071	39	00111001	9	9	Nine
58	072	3A	00111010	:	:	Colon
59	073	3B	00111011	;	;	Semicolon
60	074	3C	00111100	<	<	Less than
61	075	3D	00111101	=	=	Equals
62	076	3E	00111110	>	>	Greater than
63	077	3F	00111111	?	?	Question mark
64	100	40	01000000	@	@	At symbol
65	101	41	01000001	A	A	Uppercase A
66	102	42	01000010	B	B	Uppercase B
67	103	43	01000011	C	C	Uppercase C
68	104	44	01000100	D	D	Uppercase D
69	105	45	01000101	E	E	Uppercase E
70	106	46	01000110	F	F	Uppercase F

71	107	47	01000111	G	G	Uppercase G
72	110	48	01001000	H	H	Uppercase H
73	111	49	01001001	I	I	Uppercase I
74	112	4A	01001010	J	J	Uppercase J
75	113	4B	01001011	K	K	Uppercase K
76	114	4C	01001100	L	L	Uppercase L
77	115	4D	01001101	M	M	Uppercase M
78	116	4E	01001110	N	N	Uppercase N
79	117	4F	01001111	O	O	Uppercase O
80	120	50	01010000	P	P	Uppercase P
81	121	51	01010001	Q	Q	Uppercase Q
82	122	52	01010010	R	R	Uppercase R
83	123	53	01010011	S	S	Uppercase S
84	124	54	01010100	T	T	Uppercase T
85	125	55	01010101	U	U	Uppercase U
86	126	56	01010110	V	V	Uppercase V
87	127	57	01010111	W	W	Uppercase W
88	130	58	01011000	X	X	Uppercase X
89	131	59	01011001	Y	Y	Uppercase Y
90	132	5A	01011010	Z	Z	Uppercase Z
91	133	5B	01011011	[	[	Opening bracket

92	134	5C	01011100	\	\	Backslash
93	135	5D	01011101	]	]	Closing bracket
94	136	5E	01011110	^	^	Caret - circumflex
95	137	5F	01011111	_	_	Underscore
96	140	60	01100000	`	`	Grave accent
97	141	61	01100001	a	a	Lowercase a
98	142	62	01100010	b	b	Lowercase b
99	143	63	01100011	c	c	Lowercase c
100	144	64	01100100	d	d	Lowercase d
101	145	65	01100101	e	e	Lowercase e
102	146	66	01100110	f	f	Lowercase f
103	147	67	01100111	g	g	Lowercase g
104	150	68	01101000	h	h	Lowercase h
105	151	69	01101001	i	i	Lowercase i
106	152	6A	01101010	j	j	Lowercase j
107	153	6B	01101011	k	k	Lowercase k
108	154	6C	01101100	l	l	Lowercase l
109	155	6D	01101101	m	m	Lowercase m
110	156	6E	01101110	n	n	Lowercase n
111	157	6F	01101111	o	o	Lowercase o
112	160	70	01110000	p	p	Lowercase p

113	161	71	01110001	q	q	Lowercase q
114	162	72	01110010	r	r	Lowercase r
115	163	73	01110011	s	s	Lowercase s
116	164	74	01110100	t	t	Lowercase t
117	165	75	01110101	u	u	Lowercase u
118	166	76	01110110	v	v	Lowercase v
119	167	77	01110111	w	w	Lowercase w
120	170	78	01111000	x	x	Lowercase x
121	171	79	01111001	y	y	Lowercase y
122	172	7A	01111010	z	z	Lowercase z
123	173	7B	01111011	{	{	Opening brace
124	174	7C	01111100	|	|	Vertical bar
125	175	7D	01111101	}	}	Closing brace
126	176	7E	01111110	~	~	Equivalency sign (tilde)
127	177	7F	01111111		&#127;	Delete

Extended ASCII Codes

Below is set of additional 128 Extended ASCII Codes according to ISO 8859-1, also called ISO Latin-1.

DEC	OCT	HEX	BIN	Symbol	HTMLCode	Description
128	200	80	10000000	€	€	Euro sign
129	201	81	10000001			
130	202	82	10000010	‚	‚	Single low-9 quotation mark

131	203	83	10000011	ƒ	ƒ	Latin small letter f with hook
132	204	84	10000100	„	„	Double low-9 quotation mark
133	205	85	10000101	…	…	Horizontal ellipsis
134	206	86	10000110	†	†	Dagger
135	207	87	10000111	‡	‡	Double dagger
136	210	88	10001000	ˆ	ˆ	Modifier letter circumflex accent
137	211	89	10001001	‰	‰	Per mille sign
138	212	8A	10001010	Š	Š	Latin capital letter S with caron
139	213	8B	10001011	‹	‹	Single left-pointing angle quotation
140	214	8C	10001100	Œ	Œ	Latin capital ligature OE
141	215	8D	10001101			
142	216	8E	10001110	Ž	Ž	Latin capital letter Z with caron
143	217	8F	10001111			
144	220	90	10010000			
145	221	91	10010001	'	‘	Left single quotation mark
146	222	92	10010010	'	’	Right single quotation mark
147	223	93	10010011	"	“	Left double quotation mark
148	224	94	10010100	"	”	Right double quotation mark
149	225	95	10010101	•	•	Bullet

150	226	96	10010110	–	–	En dash
151	227	97	10010111	—	—	Em dash
152	230	98	10011000	˜	˜	Small tilde
153	231	99	10011001	™	™	Trade mark sign
154	232	9A	10011010	š	š	Latin small letter S with caron
155	233	9B	10011011	›	›	Single right-pointing angle quotation mark
156	234	9C	10011100	œ	œ	Latin small ligature oe
157	235	9D	10011101			
158	236	9E	10011110	ž	ž	Latin small letter z with caron
159	237	9F	10011111	Ÿ	Ÿ	Latin capital letter Y with diaeresis
160	240	A0	10100000			Non-breaking space
161	241	A1	10100001	¡	¡	Inverted exclamation mark
162	242	A2	10100010	¢	¢	Cent sign
163	243	A3	10100011	£	£	Pound sign
164	244	A4	10100100	¤	¤	Currency sign
165	245	A5	10100101	¥	¥	Yen sign
166	246	A6	10100110	¦	¦	Pipe, Broken vertical bar
167	247	A7	10100111	§	§	Section sign
168	250	A8	10101000	¨	¨	Spacing diaeresis - umlaut
169	251	A9	10101001	©	©	Copyright sign

170	252	AA	10101010	ª	ª	Feminine ordinal indicator
171	253	AB	10101011	«	«	Left double angle quotes
172	254	AC	10101100	¬	¬	Not sign
173	255	AD	10101101		­	Soft hyphen
174	256	AE	10101110	®	®	Registered trade mark sign
175	257	AF	10101111	¯	¯	Spacing macron - overline
176	260	B0	10110000	°	°	Degree sign
177	261	B1	10110001	±	±	Plus-or-minus sign
178	262	B2	10110010	²	²	Superscript two - squared
179	263	B3	10110011	³	³	Superscript three - cubed
180	264	B4	10110100	´	´	Acute accent - spacing acute
181	265	B5	10110101	µ	µ	Micro sign
182	266	B6	10110110	¶	¶	Pilcrow sign - paragraph sign
183	267	B7	10110111	·	·	Middle dot - Georgian comma
184	270	B8	10111000	¸	¸	Spacing cedilla
185	271	B9	10111001	¹	¹	Superscript one
186	272	BA	10111010	º	º	Masculine ordinal indicator
187	273	BB	10111011	»	»	Right double angle quotes
188	274	BC	10111100	¼	¼	Fraction one quarter
189	275	BD	10111101	½	½	Fraction one half

190	276	BE	10111110	¾	¾	Fraction three quarters
191	277	BF	10111111	¿	¿	Inverted question mark
192	300	C0	11000000	À	À	Latin capital letter A with grave
193	301	C1	11000001	Á	Á	Latin capital letter A with acute
194	302	C2	11000010	Â	Â	Latin capital letter A with circumflex
195	303	C3	11000011	Ã	Ã	Latin capital letter A with tilde
196	304	C4	11000100	Ä	Ä	Latin capital letter A with diaeresis
197	305	C5	11000101	Å	Å	Latin capital letter A with ring above
198	306	C6	11000110	Æ	Æ	Latin capital letter AE
199	307	C7	11000111	Ç	Ç	Latin capital letter C with cedilla
200	310	C8	11001000	È	È	Latin capital letter E with grave
201	311	C9	11001001	É	É	Latin capital letter E with acute
202	312	CA	11001010	Ê	Ê	Latin capital letter E with circumflex
203	313	CB	11001011	Ë	Ë	Latin capital letter E with diaeresis
204	314	CC	11001100	Ì	Ì	Latin capital letter I with grave
205	315	CD	11001101	Í	Í	Latin capital letter I with

						acute
206	316	CE	11001110	Î	Î	Latin capital letter I with circumflex
207	317	CF	11001111	Ï	Ï	Latin capital letter I with diaeresis
208	320	D0	11010000	Ð	Ð	Latin capital letter ETH
209	321	D1	11010001	Ñ	Ñ	Latin capital letter N with tilde
210	322	D2	11010010	Ò	Ò	Latin capital letter O with grave
211	323	D3	11010011	Ó	Ó	Latin capital letter O with acute
212	324	D4	11010100	Ô	Ô	Latin capital letter O with circumflex
213	325	D5	11010101	Õ	Õ	Latin capital letter O with tilde
214	326	D6	11010110	Ö	Ö	Latin capital letter O with diaeresis
215	327	D7	11010111	×	×	Multiplication sign
216	330	D8	11011000	Ø	Ø	Latin capital letter O with slash
217	331	D9	11011001	Ù	Ù	Latin capital letter U with grave
218	332	DA	11011010	Ú	Ú	Latin capital letter U with acute
219	333	DB	11011011	Û	Û	Latin capital letter U with circumflex
220	334	DC	11011100	Ü	Ü	Latin capital letter U with

						diaeresis
221	335	DD	11011101	Ý	Ý	Latin capital letter Y with acute
222	336	DE	11011110	Þ	Þ	Latin capital letter THORN
223	337	DF	11011111	ß	ß	Latin small letter sharp s - ess-zed
224	340	E0	11100000	à	à	Latin small letter a with grave
225	341	E1	11100001	á	á	Latin small letter a with acute
226	342	E2	11100010	â	â	Latin small letter a with circumflex
227	343	E3	11100011	ã	ã	Latin small letter a with tilde
228	344	E4	11100100	ä	ä	Latin small letter a with diaeresis
229	345	E5	11100101	å	å	Latin small letter a with ring above
230	346	E6	11100110	æ	æ	Latin small letter ae
231	347	E7	11100111	ç	ç	Latin small letter c with cedilla
232	350	E8	11101000	è	è	Latin small letter e with grave
233	351	E9	11101001	é	é	Latin small letter e with acute
234	352	EA	11101010	ê	ê	Latin small letter e with circumflex
235	353	EB	11101011	ë	ë	Latin small letter e with diaeresis

236	354	EC	11101100	ì	ì	Latin small letter i with grave
237	355	ED	11101101	í	í	Latin small letter i with acute
238	356	EE	11101110	î	î	Latin small letter i with circumflex
239	357	EF	11101111	ï	ï	Latin small letter i with diaeresis
240	360	F0	11110000	ð	ð	Latin small letter eth
241	361	F1	11110001	ñ	ñ	Latin small letter n with tilde
242	362	F2	11110010	ò	ò	Latin small letter o with grave
243	363	F3	11110011	ó	ó	Latin small letter o with acute
244	364	F4	11110100	ô	ô	Latin small letter o with circumflex
245	365	F5	11110101	õ	õ	Latin small letter o with tilde
246	366	F6	11110110	ö	ö	Latin small letter o with diaeresis
247	367	F7	11110111	÷	÷	Division sign
248	370	F8	11111000	ø	ø	Latin small letter o with slash
249	371	F9	11111001	ù	ù	Latin small letter u with grave
250	372	FA	11111010	ú	ú	Latin small letter u with acute
251	373	FB	11111011	û	û	Latin small letter u with circumflex

252	374	FC	11111100	ü	ü	Latin small letter u with diaeresis
253	375	FD	11111101	ý	ý	Latin small letter y with acute
254	376	FE	11111110	þ	þ	Latin small letter thorn
255	377	FF	11111111	ÿ	ÿ	Latin small letter y with diaeresis

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

chroot

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

close

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

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";
```

tutorialspoint
SIMPLYEASYLEARNING

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

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.

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

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";

}
if( defined($var2) ){

  print "var2 is also defined\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:

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:

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:

die 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:

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:

do BLOCK

do EXPR

do SUB(LIST)

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:

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

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:

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:

```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";
```

```
}

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:

Name = default

Aliases =

Addrtype = 2

Net = 0

Name = loopback

Aliases =

Addrtype = 2

Net = 2130706432

Name = link-local

Aliases =

Addrtype = 2

Net = 2851995648

Name = default

Aliases =

Addrtype = 2

Net = 0

Name = loopback

Aliases =

Addrtype = 2

Net = 2130706432

Name = link-local

Aliases =

Addrtype = 2

Net = 2851995648

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:

```perl
#!/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 databse /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 = 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

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

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:

#!/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";

```perl
   print "Protocol Name = $protocol_name\n";
}


setservent();   # Rewind the database;


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:

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 "--------------\n";

      }

      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.

Operation	Result
LOCK_SH	Set shared lock.
LOCK_EX	Set exclusive lock.
LOCK_UN	Unlock specified file.
LONG_NB	Set lock without blocking.

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:

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

fork

Description

This function forks a new process using the fork( ) system call. Any shared sockets or filehandles are duplicated across processes. You must ensure that you wait on your children to prevent "zombie" processes from forming.

Syntax

Following is the simple syntax for this function:

fork

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;

}

print "This is parent process and child ID is $pid\n";

print "Parent 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

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:

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:

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

select(STDOUT);

We would then associate EMPLOYEE with STDOUT by setting the new format name with STDOUT, using the special variable $~

$~ = "EMPLOYEE";

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

```
Kirsten          12

Mohammad            35

Suhi             15
```

Namrat   10

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 `$%` vairable along with header as follows

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.

```
format EMPLOYEE_BOTTOM =
   End of Page @<
           $%

   .
```

This will give you following result

```
------------------------

  Name            Age   Page 1

------------------------

  Kirsten         12

  Mohammad            35

  Suhi            15

  Namrat           10

  End of Page 1
```

For a complete set of variables related to formating, please refer to Perl Special Variablessection.

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

getc

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 scalr context and in list context (Name, Password, Group ID, and member list.

Example
Following is the example code showing its basic usage:

```perl
#!/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 scalr 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
```

```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

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 scalr 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";
   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

.

.

.

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 scalr context and empty list on error otherwise host record 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
   $name  = gethostbyaddr($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 scalr context and empty list on error otherwise host record in list context.

Example

Following is the example code showing its basic usage:

```perl
#!/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 scalr 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:

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:

```
#!/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:

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:

```
#!/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:

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

use Socket;

($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:

```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";
}
```

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

getpgrp

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:

getpgrp EXPR

getpgrp

Return Value

This function returns process group ID.

Example

Following is the example code showing its basic usage:

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

```perl
$pgid = getpgrp();
```

```perl
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

getprotobyname

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:

getprotobyname 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) = getprotobyname("tcp");


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

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);
```

```perl
print "Name = $name\n";
```

```perl
print "Aliases = $aliases\n";
```

```perl
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

Name = manet

Aliases = manet

Protocol Number = 138

Name = hip

Aliases = HIP

Protocol Number = 139

Name = shim6

Aliases = Shim6

Protocol Number = 140

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:

```perl
#!/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

getpwnam

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:

getpwnam 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:

($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:

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:

```perl
#!/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:

```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";
}
```

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

.

.

.

Name = iclpv-sc

Aliases =

Port Number = 1390

Protocol Name = tcp

Name = iclpv-sc

Aliases =

Port Number = 1390

Protocol Name = udp

Name = iclpv-sas

Aliases =

Port Number = 1391

Protocol Name = tcp

Name = iclpv-sas

Aliases =

Port Number = 1391

Protocol Name = udp

Na

getsockname

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:

getsockname SOCKET

Return Value

This function returns undef on error otherwise Packed address of local socket in scalar cotext.

Example

Following is the example code showing its basic usage:

When above code is executed, it produces the following result

getsockopt

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:

OPTNAME	Result
SO_DEBUG	Get status of recording of debugging information
SO_REUSEADDR	Get status of local address reuse
SO_KEEPALIVE	Get status of keep connections alive
SO_DONTROUTE	Get status of routing bypass for outgoing messages
SO_LINGER	Get status of linger on close if data is present

SO_BROADCAST   Get status of permission to transmit broadcast messages

SO_OOBINLINE   Get status of out-of-band data in band

SO_SNDBUF   Get buffer size for output

SO_RCVBUF Get buffer size for input

SO_TYPE   Get the type of the socket

SO_ERROR   Get and clear error on the socket

TCP_NODELAY   To disable the Nagle buffering algorithm.

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 = getprotobyname("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:

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:

```
#!/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:

($sec,$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
```

```perl
@weekday = ("Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat");
```

tutorialspoint
SIMPLYEASYLEARNING

$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

Formated time = 1/8/2013 9:6:41 Sun

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:

```perl
#!/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:

```perl
#!/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:

```perl
#!/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
```

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:

```perl
#!/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

int

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:

Name	Effect
SIGABRT	Aborts the process
SIGARLM	Alarm signal
SIGFPE	Arithmetic exception
SIGHUP	Hang up.
SIGILL	Illegal instruction
SIGINT	Interrupt
SIGKILL	Termination signal
SIGPIPE	Write to a pipe with no readers.
SIGQUIT	Quit signal.
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:

```perl
#!/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

lc

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:

length EXPR

length

Return Value

This function returns the size of string.

Example

Following is the example code showing its basic usage:

```
#!/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:

```
#!/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

```perl
#!/usr/bin/perl -w

# server.pl

#--------------------


use strict;

use Socket;


# use port 7890 as default

my $port = shift || 7890;

my $proto = getprotobyname('tcp');


# 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('Sn4x8', AF_INET, $port, "\0\0\0\0"))

    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(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:

local LIST

Return Value

This function does not return any value.

Example
Following is the example code showing its basic usage:

```
#!/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:

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:

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:

```
#!/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:

```
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:

```
#!/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

lstat

Return Value

This function returns a list of 13 elements in list context, these fields are as follows:

```
 0 dev      device number of filesystem
 1 ino      inode number
 2 mode     file mode  (type and permissions)
 3 nlink    number of (hard) links to the file
 4 uid      numeric user ID of file's owner
 5 gid      numeric group ID of file's owner
 6 rdev     the device identifier (special files only)
 7 size     total size of file, in bytes
 8 atime    last access time in seconds since the epoch
 9 mtime    last modify time in seconds since the epoch
10 ctime    inode change time in seconds since the epoch (*)
11 blksize  preferred block size for file system I/O
12 blocks   actual number of blocks allocated
```

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:

```
#!/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

Andrew

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:

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:

```
#!/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 Sept 08 11:55 testdir

msgctl

Description

This function calls the system function msgctrl( ) 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:

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

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:

my 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


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();
}
sub mysub
{
   print "$string\n";
}
```

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

next

Return Value

This function does not return any value.

Example

Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w

@list = (1,2,3,4,5,5,3,6,7,1 );

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:

no Module VERSION LIST

no Module VERSION

no MODULE LIST

no MODULE

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

oct

Return Value

This function returns the decimal value.

Example

Following is the example code showing its basic usage:

```
#!/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:

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:

open FILEHANDLE, EXPR, LIST

open FILEHANDLE, EXPR

open FILEHANDLE

Return Value

This function returns o 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 opend in read-only mode

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.

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

```
open(DATA, "<file.txt");
```

```
while(<DATA>)
```

```
{
```

```
  print "$_";
```

```
}
```

Following is the syntax to open file.txt in writing mode. Here less than > sign indicates that file has to be opend in writing mode

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:

open(DATA, "+<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 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.

Entities	Definition
< or r	Read Only Access
> or w	Creates, Writes, and Truncates
>> or a	Writes, Appends, and Creates
+< or r+	Reads and Writes
+> or w+	Reads, Writes, Creates, and Truncates
+>> or a+	Reads, Writes, Appends, and Creates

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:

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

ord

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.

Character	Description

a	ASCII character string padded with null characters
A	ASCII character string padded with spaces
b	String of bits, lowest first
B	String of bits, highest first
c	A signed character (range usually -128 to 127)
C	An unsigned character (usually 8 bits)
d	A double-precision floating-point number
f	A single-precision floating-point number
h	Hexadecimal string, lowest digit first
H	Hexadecimal string, highest digit first
i	A signed integer
I	An unsigned integer
l	A signed long integer
L	An unsigned long integer
n	A short integer in network order
N	A long integer in network order
p	A pointer to a string
s	A signed short integer
S	An unsigned short integer

u	Convert to uuencode format
v	A short integer in VAX (little-endian) order
V	A long integer in VAX order
x	A null byte
X	Indicates "go back one byte"
@	Fill with nulls (ASCII 0)

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";

$bits = pack( "x" );

# $bits is now a null chracter.

print "bits are $bits\n";

$bits = pack( "sai", 255, "T", 30 );

# creates a seven charcter string on most computers'

print "bits are $bits\n";


@array = unpack( "sai", "$bits" );


#Array now contains three elements: 255, T and 30.

print "Array $array[0]\n";

print "Array $array[1]\n";

print "Array $array[2]\n";
```

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

bits are A

bits are

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";

close CHILDREAD;

close PARENTWRITE;

exit;
```

}

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

pop

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:

```perl
#!/usr/bin/perl -w

$string = "That is test";
@list = (1,2,3,4,5,6,);
$index = index ($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.

Format	Result
%%	A percent sign
%c	A character with the given ASCII code
%s	A string
%d	A signed integer (decimal)
%u	An unsigned integer (decimal)
%o	An unsigned integer (octal)
%x	An unsigned integer (hexadecimal)
%X	An unsigned integer (hexadecimal using uppercase characters)
%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:

Flag	Result
space	Prefix positive number with a space
+	Prefix positive number with a plus sign
-	Left-justify within field
0	Use zeros, not spaces, to right-justify
#	Prefix non-zero octal with .0. and hexadecimal with .0x.
number	Minimum field width
.number	Specify precision (number of digits after decimal point) for floating point numbers
l	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:

printf FILEHANDLE FORMAT, LIST

printf FORMAT, LIST

tutorialspoint
SIMPLYEASYLEARNING

Return Value

This function

Example

Following is the example code showing its basic usage:

```
#!/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:

```
#!/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 retuns a single-quoted string.

Example

Following is the example code showing its basic usage:

```
#!/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:

qq

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:

qq (string)

Return Value

This function returns a double-quoted string.

Example

Following is the example code showing its basic usage:

```
#!/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*\

\[\.\]*

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:

qw EXPR

Return Value

This function return a list consisting of the element of LIST evaluated as if they were single-quoted.

Example
Following is the example code showing its basic usage:

```
#!/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

qx

Description

This function is a alternative to using back-quotes to execute system commands. For example, qx(ls -l) will execute the UNIX 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:

```
#!/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:

rand EXPR

rand

Return Value

This function returns a Floating point number.

Example

Following is the example code showing its basic usage:

```
#!/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:

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, $buffer, 100) )
{
  print("$buffer\n");
}
close(FILE);
```

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

```
kerberos_master 751/udp # Kerberos authentication
```

```
kerberos_master 751/tcp # Kerberos authentication
```

```
passwd_server  752/udp # Kerberos passwd server
```

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:

```
#!/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
```

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 equvivalent 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:

```
#!/usr/bin/perl -w


my($buffer) = "";
```

```
open(FILE, "/etc/services") or

    die("Error reading file, stopped");


$buffer = <FILE>;

print("$buffer");


$buffer = readline( *FILE );

print("$buffer");


close(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

readlink

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:

/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:

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:

```
#!/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:

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
- 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:

ref EXPR

ref

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:

```
#!/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:

```
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.

```
#!/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:

require EXPR

require

Return Value

This function does not return any value.

Example
Following is the example code showing its basic usage:

```
#!/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

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
Following is the example code showing its basic usage:

```
#!/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:

.

..

testdir

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:

rindex STR, SUBSTR, POSITION

rindex STR, SUBSTR

Return Value

This function returns undef on failure otherwise position of last occurence.

Example

Following is the example code showing its basic usage:

```perl
#!/usr/bin/perl -w


$pos = rindex("abcdefghijiklmdef", "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("abcdefghijiklmdef", "def", $pos-3 );

print "Found position of def $pos\n";
```

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

rmdir

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:

```perl
#!/usr/bin/perl -w

$string = "This is Test";
# this will replcase 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:

```perl
#!/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:

```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

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:

select FILEHANDLE

select

select RBITS, WBITS, EBITS, TIMEOUT

Return Value

TRhis 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:

```
#!/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

semctl

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:

semctl ID, SEMNUM, CMD, ARG

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, $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 $! ";
}
```

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

Right....

Left....

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:

```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 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, $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 $! ";
}
```

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....

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 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, $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()

```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 $! ";
}
```

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:

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:

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

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:

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

Members =

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() retriews 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() retriews the information fro 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:

Name = default

Aliases =

Addrtype = 2

Net = 0

Name = loopback

Aliases =

Addrtype = 2

Net = 2130706432

Name = link-local

Aliases =

Addrtype = 2

Net = 2851995648

Name = default

Aliases =

Addrtype = 2

Net = 0

Name = loopback

Aliases =

Addrtype = 2

Net = 2130706432

Name = link-local

Aliases =

Addrtype = 2

Net = 2851995648

setpgrp

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:

setpgrp 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:

```perl
#!/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() retriews 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:

```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 = rsvp-e2e-ignore

Aliases = RSVP-E2E-IGNORE

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

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:

setpwent

Return Value

This function does not return any value.

Example
Following is the example code showing its basic usage:

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

```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 databse /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 =

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

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() retriews 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:

setservent STAYOPEN

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 Tabl

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.

OPTNAME	Description
SO_DEBUG	Enable/disable recording of debugging information.
SO_REUSEADDR	Enable/disable local address reuse.
SO_KEEPALIVE	Enable/disable keep connections alive.
SO_DONTROUTE	Enable/disable routing bypass for outgoing messages.
SO_LINGER	Linger on close if data is present.
SO_BROADCAST	Enable/disable permission to transmit broadcast messages.
SO_OOBINLINE	Enable/disable reception of out-of-band data in band.
SO_SNDBUF	Set buffer size for output.
SO_RCVBUF	Set buffer size for input.
SO_TYPE	Get the type of the socket (get only).
SO_ERROR	Get and clear error on the socket (get only).

Example

Following is the example code showing its basic usage:

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

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])

shift

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.

Command	Description
IPC_STAT	Places the current value of each member of the data structure associated with ID into the scalar ARG
IPC_SET	Sets the value of the following members o of the data structure associated with ID to the corresponding values found in the packed scalar ARG
IPC_RMID	Removes the shared memory identifier specified by ID from the system and destroys the shared memory segment and data structure associated with it
SHM_LOCK	Locks the shared memory segment specified by ID in memory
SHM_UNLOCK	Unlocks the shared memory segment specified by ID

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
```

tutorialspoint
SIMPLYEASYLEARNING

```
# 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 itl
```

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 itno 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 itl
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 itno 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

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 itl
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 itno 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 itl
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 itno 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.ve 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:

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:

sin EXPR

sin

Return Value

This function returns the Floating Point sin value of EXPR

Example

Following is the example code showing its basic usage:

```
#!/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:

sleep EXPR

sleep

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:

socket

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:
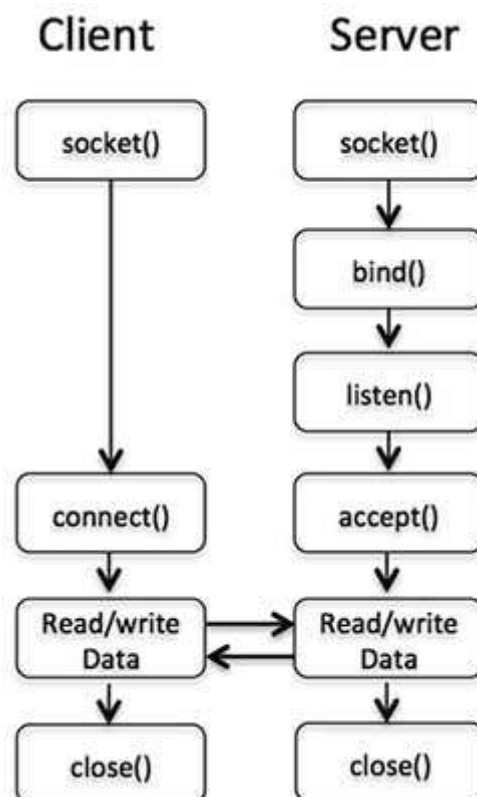
Server Side Socket Calls

The socket() call

The **socket()** call is the first call in establishing a network connection is creating a socket. This call has following syntax:

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:

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.

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:

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 ie wait for the incoming requests. This call has following syntax:

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:

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.

while(1) {

  accept(NEW_SOCKET, SOCKT);

  ....

}

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:

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.

$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

#!/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 $!\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 recieved 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,(getprotobyname('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 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:

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:

```perl
#!/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:

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:

```
#!/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:

sprintf FORMAT, LIST

Return Value

This function returns SCALAR, a formatted text string.

Example

Following is the example code showing its basic usage:

```
#!/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:

```
sqrt EXPR


sqrt
```

Return Value

This function returns the Floating point number.

Example

Following is the example code showing its basic usage:

```
#!/usr/bin/perl -w


$result = sqrt(4);

print " sqrt(4) is $result\n";
```

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:

srand EXPR

srand

Return Value

This function does not return any value.

Example
Following is the example code showing its basic usage:

```perl
#!/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: 0.811688061411591.

Here's the same random number: 0.811688061411591.

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.

Element Description

0   Device number of file system

1   Inode number

2   File mode (type and permissions)

3   Number of (hard) links to the file

4   Numeric user ID of file.s owner

5   Numeric group ID of file.s owner

6   The device identifier (special files only)

7   File size, in bytes

8   Last access time since the epoch

9   Last modify time since the epoch

10   Inode change time (not creation time!) since the epoch

11   Preferred block size for file system I/O

12   Actual number of blocks allocated

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\n");

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:

study EXPR

study

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:

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:

sub NAME PROTO ATTRS BLOCK# Named, prototype, attributes, definition

sub NAME ATTRS BLOCK # Named, attributes, definition

sub NAME PROTO BLOCK # Named, prototype, definition

sub NAME BLOCK # Named, definition

sub NAME PROTO ATTRS # Named, prototype, attributes

sub NAME ATTRS # Named, attributes

sub NAME PROTO # Named, prototype

sub NAME # Named

sub PROTO ATTRS BLOCK # Anonymous, prototype, attributes, definition

sub ATTRS BLOCK # Anonymous, attributes, definition

sub PROTO BLOCK # Anonymous, prototype, definition

sub BLOCK # Anonymous, 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:

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 valuye is $temp\n";


$temp = substr("okay", 1,2);

print "Substring valuye is $temp\n";
```

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

Substring valuye is ay

Substring valuye 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::

```
#!/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:

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:

```
#!/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:

PID of this process is 23705

sysopen

Description

This function is equivalent to the underlying C and operating system call open( ). Opens the file specified by FILENAME, associating it with FILEHANDLE. The MODE argument specifies how the file should be opened. The values of MODE are system dependent, but some values are historically set. Values of 0, 1, and 2 mean read-only, write-only, and read/write, respectively. The supported values are available in the Fcntl module, and are summarized in below Table.

Note that FILENAME is strictly a file name; no interpretation of the contents takes place (unlike open), and the mode of opening is defined by the MODE argument.

If the file has to be created, and the O_CREAT flag has been specified in MODE, then the file is created with the permissions of PERMS. The value of PERMS must be specified in traditional Unix-style hexadecimal. If PERMS is not specified, then Perl uses a default mode of 0666 (read/write on user/group/other).

Flag	Description
O_RDONLY	Read only.
O_WRONLY	Write only.
O_RDWR	Read and write.
O_CREAT	Create the file if it doesn.t already exist.
O_EXCL	Fail if the file already exists.
O_APPEND	Append to an existing file.
O_TRUNC	Truncate the file before opening.
O_NONBLOCK	Non-blocking mode.
O_NDELAY	Equivalent of O_NONBLOCK.
O_EXLOCK	Lock using flock and LOCK_EX.
O_SHLOCK	Lock using flock and LOCK_SH.

O_DIRECTOPRY   Fail if the file is not a directory.

O_NOFOLLOW   Fail if the last path component is a symbolic link.

O_BINARY   Open in binary mode (implies a call to binmode).

O_LARGEFILE   Open with large (>2GB) file support.

O_SYNC   Write data physically to the disk, instead of

write buffer.

O_NOCTTY   Don't make the terminal file being opened

the processescontrolling terminal, even if you

don.t have one yet.

Syntax

Following is the simple syntax for this function:

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.

EEK_SET -> 0

Sets the new position absolutely to POSITION bytes

within the file

SEEK_CUR -> 1

Sets the new position to the current position plus

POSITION bytes within the file

SEEK_END -> 2

Sets the new position to 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 wai

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.

Syntax

Following is the simple syntax for this function:

syswrite FILEHANDLE, SCALAR, LENGTH, OFFSET

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:

tell FILEHANDLE

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.

```
#!/usr/bin/perl -w


open( FILE, "</tmp/test.txt" ) || die "Enable 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 with in file $position\n";

close(FILE);
```

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

First Character is T

Second Character is h

Position with in 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\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

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:

```
#!/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 "Formated 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

Formated 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:

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

uc

Return Value

This function returns String in uppercase.

Example

Following is the example code showing its basic usage:

```perl
#!/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:

```perl
#!/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.

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:

undef EXPR

undef

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:

```
#!/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.

Character	Description
a	ASCII character string padded with null characters
A	ASCII character string padded with spaces
b	String of bits, lowest first
B	String of bits, highest first
c	A signed character (range usually -128 to 127)
C	An unsigned character (usually 8 bits)
d	A double-precision floating-point number
f	A single-precision floating-point number

h	Hexadecimal string, lowest digit first
H	Hexadecimal string, highest digit first
i	A signed integer
I	An unsigned integer
l	A signed long integer
L	An unsigned long integer
n	A short integer in network order
N	A long integer in network order
p	A pointer to a string
s	A signed short integer
S	An unsigned short integer
u	Convert to uuencode format
v	A short integer in VAX (little-endian) order
V	A long integer in VAX order
x	A null byte
X	Indicates "go back one byte"
@	Fill with nulls (ASCII 0)

Example

Following is the example code showing its basic usage:

```
#!/usr/bin/perl -w
```

```
$bits = pack("c", 65);

# prints A, which is ASCII 65.

print "bits are $bits\n";

$bits = pack( "x" );

# $bits is now a null chracter.

print "bits are $bits\n";

$bits = pack( "sai", 255, "T", 30 );

# creates a seven charcter string on most computers'

print "bits are $bits\n";


@array = unpack( "sai", "$bits" );


#Array now contains three elements: 255, A and 47.

print "Array $array[0]\n";

print "Array $array[1]\n";

print "Array $array[2]\n";
```

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:

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
```

```perl
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:

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:

```
 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:

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
```

```perl
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:

values HASH

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
```

```perl
%hash = ('One' => 1,

        'Two' => 2,

        'Three' => 3,

        'Four' => 4);
```

```perl
@values = values( %hash );
```

```perl
print("Values are  ", join("-", @values), "\n");
```

```perl
@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:

```
#!/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:

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:

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:

A, B, C

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:

warn LIST

Return Value

This function does not return any value.

Example

Following is the example code showing its basic usage:

```
#!/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:

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

-r   File is readable by effective uid/gid.

-w File is writable by effective uid/gid.

-x   File is executable by effective uid/gid.

-o   File is owned by effective uid.

-R   File is readable by real uid/gid.

-W File is writable by real uid/gid.

-X   File is executable by real uid/gid.

-O   File is owned by real uid.

-e   File exists.

-z   File has zero size (is empty).

-s File has nonzero size (returns size in bytes).

-f File is a plain file.

-d File is a directory.

-l File is a symbolic link.

-p File is a named pipe (FIFO), or Filehandle is a pipe.

-S File is a socket.

-b File is a block special file.

-c File is a character special file.

-t Filehandle is opened to a tty.

-u File has setuid bit set.

-g File has setgid bit set.

-k File has sticky bit set.

-T File is an ASCII text file (heuristic guess).

-B File is a "binary" file (opposite of -T).

-M Script start time minus file modification time, in days.

-A Same for access time.

-C Same for inode change time

Example

Try out following example with some file.

```
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:

y/SEARCHLIST/REPLACEMENTLIST/

Return Value

This function returns the number of characters modified.

Example
Following is the example code showing its basic usage:

```
#!/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.