PERL PROGRAMMING

programming language

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

About the Tutorial

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

This tutorial provides a complete understanding on Perl.

Audience

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

Prerequisites

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

Copyright & Disclaimer

Table of Contents

Part 1: Perl – Basics

1. PERL – 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.

2. PERL – ENVIRONMENT

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.

3. PERL – SYNTAX OVERVIEW

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 let's 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.

4. PERL – DATA TYPES

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. PERL – 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.

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:

```
#!/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:

```
#!/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.

6. PERL – SCALARS

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:

```
#!/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:

```
#!/usr/bin/perl

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

```
# 377 octal, same as 255 decimal
$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:

```
#!/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.

```
#!/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:

```
#!/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:

```
#!/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. PERL – 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:

```
#!/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";
```