Unit-3

Perl Scripting Introduction to Perl Scripting, working with Simple Values, Lists and Hashes, Loops and Decisions, Regular Expressions, Files and Data in Perl Scripting, References &Subroutines, Running and Debugging Perl, Modules, Object-Oriented Perl.

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 is 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.

Try it Option Online

We have set up the Perl Programming environment online, so that you can compile and execute all the available examples online. It gives you confidence in what you are reading and enables you to verify the programs with different options. Feel free to modify any example and execute it online.

Try the following example using our online compiler available atCodingGround

```
#
# Hello World Program in Perl
#
print "Hello World!\n";
```

For most of the examples given in this tutorial, you will find a Try it option in our website code sections at the top right corner that will take you to the online compiler. So just make use of it and enjoy your learning.

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 Windowshttp://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 are 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 the actual 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";
```

This will produce the following result −

```
Hello, world
Hello, world
```

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";
```

This will produce the following result −

```
Hello, world
```

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, where as 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 document, 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	200.0
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
```

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. Otherwise 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 −

```
#!/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 = ☺
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 can not be interpolated

print "__FILE__ __LINE__ __PACKAGE__\n";
```

This will produce the following result −

```
File name main.pl
Line Number 4
Package main
__FILE__ __LINE__ __PACKAGE__
```

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

Here is a simple example of using the array variables −

```
#!/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. Otherwise 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.

```
#!/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 one-based indexing. It is recommended not to use any other indexing other than zero. However, let's take one example to show the usage of $[variable −

```
#!/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 −

```
#!/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 −

```
$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 −

```
%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 −

```
%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 −

```
#!/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 −

```
#!/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 −

```
#!/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 −

```
#!/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;
```

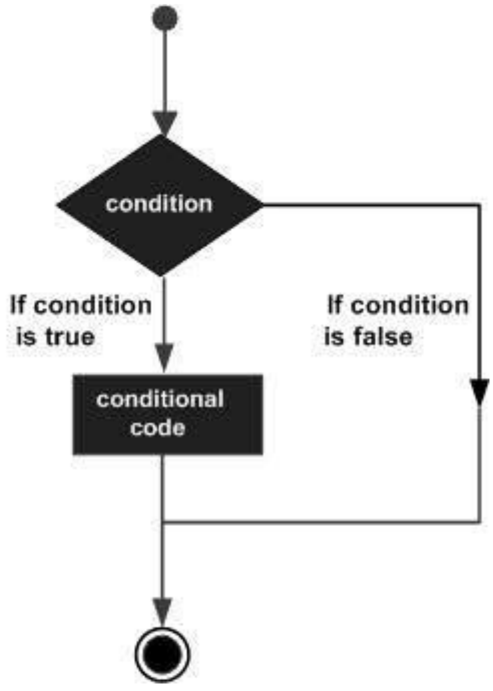
```
$data{'Ali'} = 55;
@keys = keys %data;
$size = @keys;
print "2 - Hash size:  is $size\n";

# 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.

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";
```

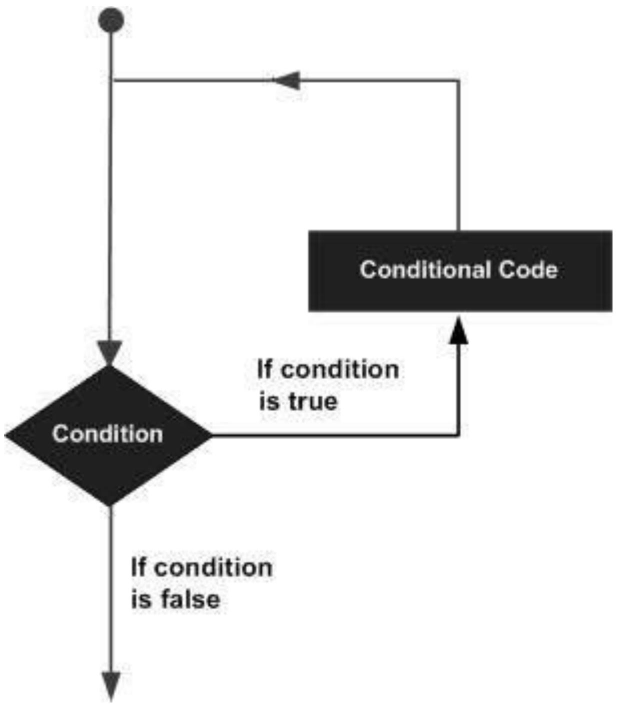
This will produce the following result −

```
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.

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.

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 −

[Show Example]

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

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 −

[Show Example]

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.

Below is a list of equity operators. Assume variable $a holds "abc" and variable $b holds "xyz" then, lets check following string equality operators:

[Show Example]

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.

Perl Assignment Operators

Assume variable $a holds 10 and variable $b holds 20, then −

[Show Example]

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 operand	$c -= $a is equivalent to $c = $c - $a
*=	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

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

[Show Example]

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	($a & $b) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either	($a | $b) will give 61

	operand.	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

Perl Logical Operators

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

[Show Example]

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.

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.

[Show Example]

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\`

Miscellaneous Operators

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

[Show Example]

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.

Perl Operators Precedence

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

[Show Example]

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
```

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 −

```
#!/usr/bin/perl


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


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


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


# Function call
Average(10, 20, 30);
```

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

```
Average for the given numbers : 20
```

Passing Lists to Subroutines

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

```
#!/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 −

```
#!/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.

Perl - 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 −

```
#!/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 −

```
#!/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 - 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 right 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 −

```
#!/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 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 peoples 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

Object Oriented Programming in PERL

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
{
   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 –

```
sub getFirstName {
   return $self->{_firstName};
}
```

Another helper function to set person's first name –

```
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.

```
#!/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;
}
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 −

```
#!/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.

```
#!/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.

```
#!/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 );

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
```