First Perl Program

Interactive Mode Programming

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

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

This execution will produce the following result –

```
Hello, world
```

Script Mode Programming

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

```
#!/usr/bin/perl

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

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

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

This execution will produce the following result –

```
Hello, world
```

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

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

Perl File Extension

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

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

Comments in Perl

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

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

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

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

```
#!/usr/bin/perl

# This is a single line comment
print "Hello, world\n";

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

This will produce the following result –

```
Hello, world
```

Whitespaces in Perl

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

```
#!/usr/bin/perl

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

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

```
#!/usr/bin/perl

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

This will produce the following result –

```
Hello
        world
```

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

Single and Double Quotes in Perl

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

```
#!/usr/bin/perl

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

This will produce the following result –

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

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

```
#!/usr/bin/perl

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

This will produce the following result –

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

"Here" Documents

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

An identifier may be either a bare word or some quoted text like we used EOF below. If identifier is quoted, the type of quote you use determines the treatment of the text inside the here 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 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.

Sr.No.	Types & Description
1	**Scalar** Scalars are simple variables. They are preceded by a dollar sign ($). A scalar is either a number, a string, or a reference. A reference is actually an address of a variable, which we will see in the upcoming chapters.
2	**Arrays** Arrays are ordered lists of scalars that you access with a numeric index, which starts with 0. They are preceded by an "at" sign (@).
3	**Hashes** Hashes are unordered sets of key/value pairs that you access using the keys as subscripts. They are preceded by a percent sign (%).

Numeric Literals

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

Type	Value
Integer	1234
Negative integer	-100
Floating point	2000
Scientific notation	16.12E14
Hexadecimal	0xffff
Octal	0577

String Literals

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

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

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

Escape sequence	Meaning
\\	Backslash
\'	Single quote
\"	Double quote
\a	Alert or bell
\b	Backspace

\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\0nn	Creates Octal formatted numbers
\xnn	Creates Hexideciamal formatted numbers
\cX	Controls characters, x may be any character
\u	Forces next character to uppercase
\l	Forces next character to lowercase
\U	Forces all following characters to uppercase
\L	Forces all following characters to lowercase
\Q	Backslash all following non-alphanumeric characters
\E	End \U, \L, or \Q

Example

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

```
#!/usr/bin/perl

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

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

# Only W will become upper case.
$str = "\uwelcome to tutorialspoint.com!";
print "$str\n";

# Whole line will become capital.
```

```
$str = "\UWelcome to tutorialspoint.com!";
print "$str\n";

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

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

This will produce the following result –

```
Welcome to
tutorialspoint.com!
Welcome to \ntutorialspoint.com!
Welcome to tutorialspoint.com!
WELCOME TO TUTORIALSPOINT.COM!
Welcome to TUTORIALSPOINT.com!
Welcome\ to\ tutorialspoint\'s\ family
```

Variables are the reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

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

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

- Scalars
- Arrays
- Hashes

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

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

Creating Variables

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

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

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

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

Here 25, "John Paul" and 1445.50 are the values assigned to *$age*, *$name* and *$salary* variables, respectively. Shortly we will see how we can assign values to arrays and hashes.

Scalar Variables

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

Here is a simple example of using scalar variables –

```
#!/usr/bin/perl

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

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

This will produce the following result –

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

Array Variables

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

Here is a simple example of using array variables –

```
#!/usr/bin/perl

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

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

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

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

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 –

Live Demo

```
#!/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 –

Sr.No.	Context & 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 "mul = $mul\n";
print "mix = $mix\n";
```

This will produce the following result –

```
str = helloworld
num = 15
mul = 20
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 main.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 hello.pl
Line Number 4
Package main
__FILE__ __LINE__ __PACKAGE__
```

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

Here is a simple example of using the array variables –

```
#!/usr/bin/perl

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

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

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

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

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

Array Creation

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

```
@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 –

Live Demo

```
#!/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 –

```
#!/usr/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.

Sr.No.	Types & 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 on-based indexing. It is recommended not to use any other indexing other than zero. However, let's take one example to show the usage of $[variable –

Live Demo

```
#!/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";
```

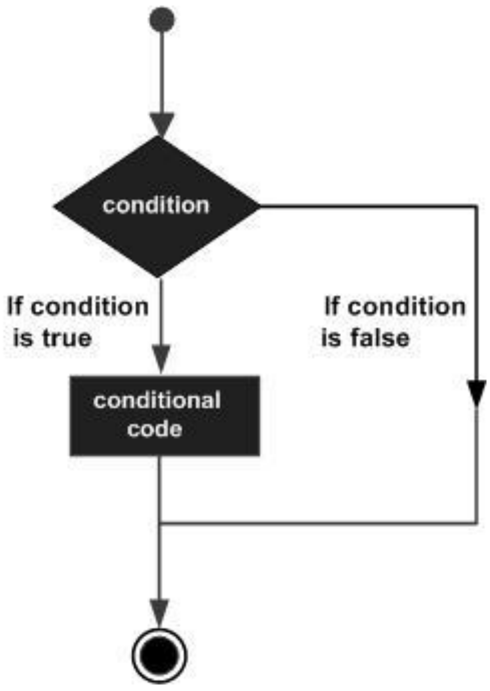
This will produce the following result –

```
1 - Hash size: is 3
2 - Hash size: is 4
3 - Hash size: is 3
```

IF-ELSE

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.

Sr.No.	Statement & Description
1	if statement An **if statement** consists of a boolean expression followed by one or more statements.
2	if...else statement An **if statement** can be followed by an optional **else statement**.
3	if...elsif...else statement An **if statement** can be followed by an optional **elsif statement** and then by an optional **else statement**.
4	unless statement An **unless statement** consists of a boolean expression followed by one or more statements.
5	unless...else statement An **unless statement** can be followed by an optional **else statement**.
6	unless...elsif..else statement An **unless statement** can be followed by an optional **elsif statement** and then by an optional **else statement**.
7	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 –

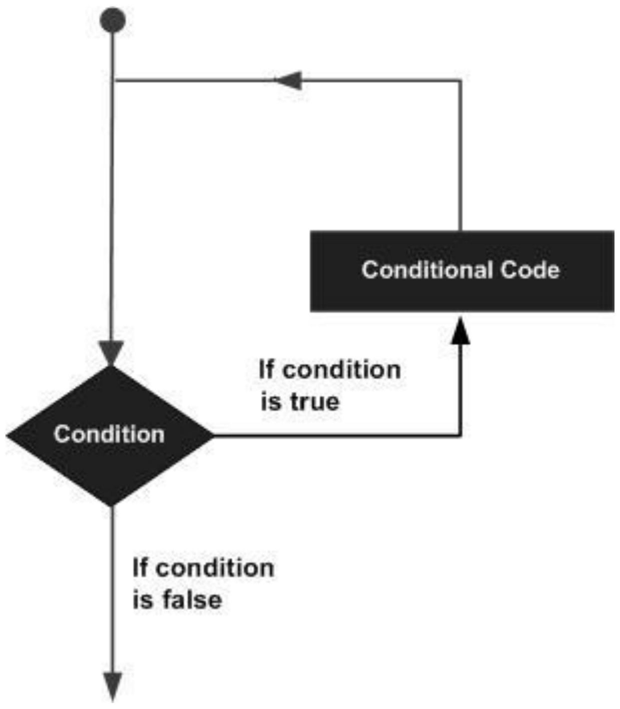
```
Ali is - Not a senior citizen
```

LOOPS

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.

Sr.No.	Loop Type & Description
1	while loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	until loop Repeats a statement or group of statements until a given condition becomes true. It tests the condition before executing the loop body.
3	for loop Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
4	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.
5	do...while loop

	Like a while statement, except that it tests the condition at the end of the loop body
6	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.

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

Sr.No.	Control Statement & Description
1	next statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
2	last statement Terminates the loop statement and transfers execution to the statement immediately following the loop.
3	continue statement A continue BLOCK, it is always executed just before the conditional is about to be evaluated again.
4	redo statement The redo command restarts the loop block without evaluating the conditional again. The continue block, if any, is not executed.
5	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 following are the Perl arithmatic operators –

Sr.No.	Operator & Description
1	**+ (Addition)** Adds values on either side of the operator **Example** – $a + $b will give 30
2	**- (Subtraction)** Subtracts right hand operand from left hand operand **Example** – $a - $b will give -10
3	*** (Multiplication)** Multiplies values on either side of the operator **Example** – $a * $b will give 200
4	**/ (Division)** Divides left hand operand by right hand operand **Example** – $b / $a will give 2
5	**% (Modulus)** Divides left hand operand by right hand operand and returns remainder **Example** – $b % $a will give 0
6	**** (Exponent)** Performs exponential (power) calculation on operators **Example** – $a**$b will give 10 to the power 20

Example

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

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

$a = 21;
$b = 10;

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

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

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

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

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

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

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

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

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

Perl Equality Operators

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

Sr.No.	Operator & Description
1	**== (equal to)** Checks if the value of two operands are equal or not, if yes then condition becomes true. **Example** – ($a == $b) is not true.
2	**!= (not equal to)** Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. **Example** – ($a != $b) is true.
3	**<=>** Checks if the value of two operands are equal or not, and returns -1, 0, or 1 depending on whether the left argument is numerically less than, equal to, or greater than the right argument. **Example** – ($a <=> $b) returns -1.
4	**> (greater than)** Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. **Example** – ($a > $b) is not true.
5	**< (less than)** Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. **Example** – ($a < $b) is true.
6	**>= (greater than or equal to)**

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

Example

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

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

$a = 21;
$b = 10;

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

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

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

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

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

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

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

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

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

```
Value of $a = 21 and value of $b = 10
$a == $b is not true
$a != $b is true
$a <=> $b returns 1
$a > $b is true
$a >= $b is true
$a < $b is not true
$a <= $b is not true
```

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

Sr.No.	Operator & Description
1	**lt** Returns true if the left argument is stringwise less than the right argument. **Example** – ($a lt $b) is true.
2	**gt** Returns true if the left argument is stringwise greater than the right argument. **Example** – ($a gt $b) is false.

3	**le** Returns true if the left argument is stringwise less than or equal to the right argument. **Example** – ($a le $b) is true.
4	**ge** Returns true if the left argument is stringwise greater than or equal to the right argument. **Example** – ($a ge $b) is false.
5	**eq** Returns true if the left argument is stringwise equal to the right argument. **Example** – ($a eq $b) is false.
6	**ne** Returns true if the left argument is stringwise not equal to the right argument. **Example** – ($a ne $b) is true.
7	**cmp** Returns -1, 0, or 1 depending on whether the left argument is stringwise less than, equal to, or greater than the right argument. **Example** – ($a cmp $b) is -1.

Example

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

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

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

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

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

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

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

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

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

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

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

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

Perl Assignment Operators

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

Sr.No.	Operator & Description
1	**=** Simple assignment operator, Assigns values from right side operands to left side operand **Example** – $c = $a + $b will assigned value of $a + $b into $c
2	**+=** Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand **Example** – $c += $a is equivalent to $c = $c + $a
3	**-=** Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand **Example** – $c -= $a is equivalent to $c = $c - $a
4	***=** Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand **Example** – $c *= $a is equivalent to $c = $c * $a
5	**/=** Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand **Example** – $c /= $a is equivalent to $c = $c / $a

6	**%=** Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand **Example** – $c %= $a is equivalent to $c = $c % a
7	****=** Exponent AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand **Example** – $c **= $a is equivalent to $c = $c ** $a

Example

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

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

$a = 10;
$b = 20;

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

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

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

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

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

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

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

$c = 2;
$a = 4;
```

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

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

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

Perl Bitwise Operators

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

$a = 0011 1100

$b = 0000 1101

$a&$b = 0000 1100

$a|$b = 0011 1101

$a^$b = 0011 0001

~$a = 1100 0011

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

Sr.No.	Operator & Description
1	**&** Binary AND Operator copies a bit to the result if it exists in both operands. **Example** – ($a & $b) will give 12 which is 0000 1100

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

Example

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

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

use integer;

$a = 60;
$b = 13;
```

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

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

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

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

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

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

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

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

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

Perl Logical Operators

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

Sr.No.	Operator & Description
1	**and** Called Logical AND operator. If both the operands are true then then condition becomes true. **Example** – ($a and $b) is false.

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

Example

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

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

$a = true;
$b = false;

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

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

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

$c = ($a or $b);
```

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

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

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

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

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

Quote-like Operators

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

Sr.No.	Operator & Description
1	**q{ }** Encloses a string with-in single quotes **Example** – q{abcd} gives 'abcd'
2	**qq{ }** Encloses a string with-in double quotes **Example** – qq{abcd} gives "abcd"
3	**qx{ }** Encloses a string with-in invert quotes **Example** – qx{abcd} gives `abcd`

Example

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

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

$a = 10;

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

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

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

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

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

Miscellaneous Operators

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

Sr.No.	Operator & Description
1	**.** Binary operator dot (.) concatenates two strings. **Example** – If $a = "abc", $b = "def" then $a.$b will give "abcdef"
2	**x**

	The repetition operator x returns a string consisting of the left operand repeated the number of times specified by the right operand. **Example** – ('-' x 3) will give ---.
3	**..** The range operator .. returns a list of values counting (up by ones) from the left value to the right value **Example** – (2..5) will give (2, 3, 4, 5)
4	**++** Auto Increment operator increases integer value by one **Example** – $a++ will give 11
5	**--** Auto Decrement operator decreases integer value by one **Example** – $a-- will give 9
6	**->** The arrow operator is mostly used in dereferencing a method or variable from an object or a class name **Example** – $obj->$a is an example to access variable $a from object $obj.

Example

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

Live Demo

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

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

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

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

$c = "-" x 3;
print "Value of \"-\" x 3 = $c\n";

@c = (2..5);
print "Value of (2..5) = @c\n";

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

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

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

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

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

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        , =>
nonassoclist operators (rightward)
right       not
left        and
left        or xor
```

Previous Page