Perl

Hos Es

Table of Contents

Hello World!

```
#!/bin/perl
use warnings;
print("Hello world!\n");
```

`use warnings;` is called *pragma* in Perl. This pragma instructs Perl to turn on additional warning reporting.

1. Perl Syntax

Basic **Perl syntax** to get started with Perl language quickly including variables, expressions, statements, block, comments, whitespaces, and keywords.

1.1. Values and Variables

You develop Perl programs to manipulate some kinds of data. The data can be either numbers, strings, or more complex such as a list. Data is held as value.

Some values

```
10
20.2
"Perl Syntax"
```

To hold a piece of data, you need **variables**. You use a variable to store a value. And through the name of the variable, you can process the value.

The following illustrates some variables in Perl:

Define variables

```
$x = 10;
$y = 20;
$s = "Perl string";
```

We have two integer variables (`$x` and `$y`) and one string variable (`$s`). For more information on Perl variables, check it out the Perl variables.

1.2. Expressions

In Perl, an expression is anything that returns a value.

The expression can be used in a larger expression or a statement. The expression can be a literal number, complex expression with operators, or a function (aka subroutine), call.

For example, 3 is an expression that returns a value of 3. The `$a + $b` is an expression that returns the sum of two variables: `$a` and `$b`.

1.3. Statements

A statement is made up of expressions. A statement is executed by Perl at run-time.

Each Perl statement must end with a semicolon (`;`). The following example shows the statements in Perl:

```
$c = $a + $b;
```

```
print($c);
```

1.4. Blocks

A block is made up of statements wrapped in curly braces `{}`. You use blocks to organize statements in the program.

The following example illustrates a block in Perl:

```
{
    $a = 1;
    $a = $a + 1;
    print($a);
}
```

Any variable declared inside a block has its own scope.

It means the variables declared inside a block only last as long as the block is executed.

1.5. Comments

In Perl, a comment begins with a hash (#) character. Perl interpreter ignores comments at both compile-time and runtime.

Typically, you use comments to document the logic of your code. The code tells you what it does however comments provides information on why the code does so.

Comments are very important and useful to you as a programmer in order to understand the code later. They're also useful to other programmers who will read and maintain your programs in the future.

Let's take a look at the following example:

```
$salary = $salary + 1.05;
```

What the code does is to increase the value of the variable `$salary` 5%. However, why it does so was not documented.

Therefore the following code with comment is much clearer.

```
# increase salary %5 for employees who achieve KPI
$salary = $salary + 1.05;
```

Perl also allows you to place a comment on the same line as the statement. See the following example:

```
$counter = 0; # reset the counter
```

It is important to use comments properly to make your code easier to understand.

1.6. Whitespace

Whitespaces are spaces, tabs, and newlines. Perl is very flexible in terms of whitespaces usages. Consider the following example:

```
$x = 20;
$y=20;
```

Both lines of code work perfectly. We surrounded the assignment operator (=) with whitespace in the first statement, but not in the second one.

Perl really doesn't care about the whitespace. However, it is a good practice to use whitespace to make the code more readable.

1.7. Keywords

Perl has a set of keywords that have special meanings to its language.

Perl keywords fall into some categories such as built-in function and control keywords.

You should always avoid using keywords to name variables, functions, modules, and other objects. Check it out the Perl keywords.

Sometimes, it is fine to use a variable name such as `$print`, which is similar to the built-in `print()` function. However, this may lead to confusion. In addition, if the program has an issue, it's more difficult to troubleshoot.

2. Perl Variables

To manipulate data in your program, you use variables.

Perl provides three types of variables: scalars, lists, and hashes to help you manipulate the corresponding data types including scalars, lists, and hashes.

We'll focus on the scalar variable in this section.

2.1. Naming variables

You use scalar variables to manipulate scalar data such as numbers and strings,

A scalar variable starts with a dollar sign ($), followed by a letter or underscore, after that, any combination of numbers, letters, and underscores. The name of a variable can be up to 255 characters.

Perl is case-sensitive. The `$variable` and `$Variable` are different variables.

Perl uses the dollar sign ($) as a prefix for the scalar variables because of the $ looks like the character S in the scalar. You can use this tip to remember when you want to declare a scalar variable.

Valid variables:

```
$gate = 10;
$_port = 20;
```

Invalid variables:

```
$4whatever = 20; # no letter or underscore found after dollar sign ($)
$email-address = "zen@example.com"; # special character (-) found
$home url = "http://localhost/perltutorial"; # space is not allowed
```

2.2. Declaring variables

Perl doesn't require you to declare a variable before using it.

For example, you can introduce a variable in your program and use it right away as follows:

```
$a = 10;
$b = 20;
$c = $a + $b;
print($c);
```

In some cases, using a variable without declaring it explicitly may lead to problems. Let's take a look at the following example:

```
$color = 'red';
```

```
print "Your favorite color is " . $colour . "\n";
```

The expected output was `Your favorite color is red`.

However, in this case, you got `Your favorite color is`, because the `$color` and `$colour` are different variables. The mistake was made because of the *different variable names*.

To prevent such cases, Perl provides a *pragma* called `strict` that requires you to declare variable explicitly before using it.

In this case, if you use the `my` keyword to declare a variable and try to run the script, Perl will issue an error message indicating that a compilation error occurred due to the `$colour` variable must be declared explicitly.

```
#!/usr/bin/perl
use strict;
my $color = 'red';
print "Your favorite color is " . $colour . "\n"
```

A variable declared with the `my` keyword is a *lexically scoped* variable.

It means the variable is only accessible inside the enclosing block or all blocks nested inside the enclosing block. In other words, the variable is local to the enclosing block.

Now, you'll learn a very important concept in programming called variable scopes.

2.3. Perl variable scopes

Let's take a look at the following example:

```
#!/usr/bin/perl
use warnings;
$color = 'red';
print("my favorite #1 color is " . $color . "\n");
# another block
{
    my $color = 'blue';
    print("my favorite #2 color is " . $color . "\n");
}
# for checking
print("my favorite #1 color is " . $color . "\n");
```

In the example above

- First, declared a global variable named `$color`
- Then, displayed the favorite color by referring to the `$color` variable. As expected, we get the red color in this case
- Next, created a new block and declared a variable with the same name `$color` using

the `my` keyword. The `$color` variable is lexical. It is a local variable and only visible inside the enclosing block

- After that, inside the block, we displayed the favorite color and we got the `blue` color. The local variable takes priority in this case
- Finally, following the block, we referred to the `$color` variable and Perl referred to the `$color` global variable

If you want to declare global variables that are visible throughout your program or from external packages, you can use `our` keyword as shown in the following code:

```
our $color = 'red';
```

2.4. Perl variable interpolation

Perl interpolates variables in double-quoted strings. It means if you place a variable inside a double-quoted string, you'll get the value of the variable instead of its name.

Let's take a look at the following example:

```
#!/usr/bin/perl
use strict;
use warnings;

my $amount = 20;
my $s = "The amount is $amount\n";
print($s);
```

Perl interpolates the value of `$amount` into the string which is 20.

Perl only interpolates scalar variables and array, not hashes. In addition, the interpolation is only applied to the double-quoted string, but not the single-quoted string

3. Perl Numbers

Perl has two kinds of numbers: integer and floating-point numbers.

3.1. Perl integers

Integers are whole numbers that have no digits after the decimal points i.e `10`, `-20` or `100`.

In Perl, integers are often expressed as decimal integers, base 10. The following illustrates some integer numbers:

```
#!/usr/bin/perl
use warnings;
use strict;

$x = 20;
$y = 100;
$z = -200;
```

When the integer number is big, you often use a comma as a separator to make it easier to read e.g., 123,763,213.

However, Perl already uses a comma (`,`) as a separator in the list so for integer numbers Perl uses an underscore character (_) instead. In this case, `123,763,213` is written in Perl as `123_763_213`.

Take a look at the following example:

```
my $a = 123_763_213;
print($a, "\n"); # 123763213
```

In the output of the example above, Perl uses no comma or underscore as the separator.

Besides the decimal format, Perl also supports other formats including binary, octal, and hexadecimal.

The following table shows you prefixes for formatting with binary, octal, and hexadecimal integers.

Number	Format
0b123	Binary integer using a prefix of 0b
0255	Octal integer using a prefix of 0
0xABC	Hexadecimal integer using a prefix of 0x

All the following integer numbers are 12 in Perl:

```
12
```

```
0b1100
014
0xC
```

3.2. Perl floating-point numbers

You use floating-point numbers to store real numbers. Perl represents floating-point numbers in two forms:

- **Fixed point**: the decimal point is fixed in the number to denote fractional part starts e.g., `100.25`
- **Scientific**: consists of a significand with the actual number value and an exponent representing the power of 10 that the significand is multiplied by, for example, `+1.0025e2` or `-1.0025E2` is `100.25.`

The floating-point number holds 8 bytes, with 11 bits reserved for the exponent and 53 bits for significand.

The range of floating-point numbers is essentially determined by the standard C library of the underlying platform where Perl is running.

4. Perl String

Perl's built-in string functions to manipulate strings.

4.1. Introduction to Perl strings

In Perl, a string is a sequence of characters surrounded by some kind of quotation marks. A string can contain ASCII, UNICODE, and escape sequences characters such as `\n`.

A Perl string has a length that depends on the amount of memory in your system, which is theoretically unlimited.

The following example demonstrates single and double-quoted strings.

```
my $s1 = "string with doubled-quotes";
my $s2 = 'string with single quote';
```

It is important to remember that the double-quoted string replaces variables inside it by their values, while the single-quoted string treats them as text. This is known as variable interpolation in Perl.

4.2. Perl string alternative delimiters

Besides the single and double quotes, Perl also allows you to use quote-like operators such as:

- The `q//` acts like a single-quoted string.
- The `qq//` acts like double-quoted string

You can choose any non-alphabetic and non-numeric characters as the delimiters, not only just characters `//`.

```
#!/usr/bin/perl
use warnings;
use strict;

my $s= q/"Are you learning Perl String today?" We asked./;
print($s ,"\n");

my $name = 'Jack';
my $s2 = qq/"Are you learning Perl String today?"$name asked./;
print($s2 ,"\n");
```

How it works

- First, defined a single-quoted string variable with the quote-like operator `q/`. The string `$s` ends with `/`
- Second, defined a double-quoted string with the quote-like operator `qq/`. In this case, we used the `$name` variable inside a string and it is replaced by its value, `Jack`

The following example demonstrates string with the ^ delimiter.

```
#!/usr/bin/perl
use warnings;
use strict;

my $s = q^A string with different delimiter ^;
print($s,"\n");
```

4.3. Perl string functions

Perl provides a set of functions that allow you to manipulate strings effectively. We cover the most commonly used string functions in the following section for your reference.

Perl string length

To find the number of characters in a string, you use the `length()` function.

```
my $s = "This is a string\n";
print(length($s),"\n"); #17
```

Changing cases of string

To change the cases of a string you use a pair of functions `lc()` and `uc()` that returns the lowercase and uppercase versions of a string.

```
my $s = "Change cases of a string\n";
print("To upper case:\n");
print(uc($s),"\n");

print("To lower case:\n");
print(lc($s),"\n");
```

Search for a substring inside a string

To search for a substring inside a string, you use `index()` and `rindex()` functions.

- The `index()` function searches for a substring inside a string from a specified position and returns the position of the first occurrence of the substring in the searched string. If the position is omitted, it searches from the beginning of the string.
- The `rindex()` function works like the `index()` function except it searches from the end of the string instead of from the beginning.

The following example demonstrates how to use the `index()` and `rindex()` functions:

```
#!/usr/bin/perl
use warnings;
use strict;

my $s = "Learning Perl is easy\n";
```

```
my $sub = "Perl";
my $p = index($s,$sub); # rindex($s,$sub);
print(qq\The substring "$sub" found at position "$p" in string "$s"\,"\n");
```

Get or modify substring inside a string

To extract a substring out of a string, you use the `substr()` function.

```
#!/usr/bin/perl
use warnings;
use strict;
# extract substring
my $s = "Green is my favorite color";
my $color  = substr($s, 0, 5);      # Green
my $end    = substr($s, -5);        # color

print($end,":",$color,"\n");

# replace substring
substr($s, 0, 5, "Red"); #Red is my favorite color
print($s,"\n");
```

Other useful Perl string functions

The following table illustrates other useful Perl string functions with their descriptions:

Function	Description
`chr`	Return ASCII or UNICODE character of a number
`crypt`	Encrypts passwords in one way fashion
`hex`	Converts a hexadecimal string to the corresponding value
`index`	Searches for a substring inside a string returns position where the first occurrence of the substring found
`lc`	Returns a lowercase version of the string
`length`	Returns the number of characters of a string
`oct`	Converts a string to an octal number
`ord`	Returns the numeric value of the first character of a string
`q/string/`	Creates single-quoted strings
`qq/string/`	Creates double-quoted strings
`reverse`	Reverses a string
`rindex`	Searches for a substring from right to left
`sprintf`	Formats string to be used with print()

Function	Description
`substr`	Gets or modifies a substring in a string
`uc`	Returns the uppercase version of the string

5. Perl Operator

Perl operators including numeric operators, string operators, and logical operators.

5.1. Numeric operators

Perl provides numeric operators to help you operate on numbers including arithmetic, Boolean and bitwise operations. Let's examine the different kinds of operators in more detail.

Arithmetic operators

Perl arithmetic operators deal with basic math such as adding, subtracting, multiplying, diving, etc. To add (`+`) or subtract (`-`) numbers, you would do something as follows:

```
#!/usr/bin/perl
use warnings;
use strict;

print 10 + 20, "\n"; # 30
print 20 - 10, "\n"; # 10
```

To multiply or divide numbers, you use divide (`/`) and multiply (`*`) operators as follows:

```
#!/usr/bin/perl
use warnings;
use strict;

print 10 * 20, "\n"; # 200
print 20 / 10, "\n"; # 2
```

When you combine adding, subtracting, multiplying, and dividing operators together, Perl will perform the calculation in an order, which is known as operator precedence.

The multiply and divide operators have higher precedence than add and subtract operators, therefore, Perl performs multiplying and dividing before adding and subtracting. See the following example:

```
print 10 + 20/2 - 5 * 2 , "\n"; # 10
```

Perl performs 20/2 and 5*2 first, therefore you will get 10 + 10 – 10 = 10.

You can use brackets `()` to force Perl to perform calculations based on the precedence you want as shown in the following example:

```
print (((10 + 20) / 2 - 5) * 2); # 20;
```

To raise one number to the power of another number, you use the exponentiation operator.

Exponentiation operators:

```
#!/usr/bin/perl
use warnings;
use strict;

print 2**3, "\n"; # = 2 * 2 * 2 = 8.
print 3**4, "\n"; # = 3 * 3 * 3 * 3 = 81.
```

To get the remainder of the division of one number by another, you use the modulo operator (%).

It is handy to use the modulo operator (%) to check if a number is odd or even by dividing it by 2 to get the remainder. If the remainder is zero, the number is even, otherwise, the number is odd. See the following example:

```
#!/usr/bin/perl
use warnings;
use strict;

print 4 % 2, "\n"; # 0 even
print 5 % 2, "\n"; # 1 odd
```

Bitwise Operators

Bitwise operators allow you to operate on numbers one bit at a time. Think of a number as a series of bits e.g., `125` can be represented in binary form as `1111101`. Perl provides all basic bitwise operators including and (`&`), or (`|`), exclusive or (`^`) , not (`~`) operators, shift right (`>>`), and shift left (`<<`) operators.

The bitwise operators perform from right to left. In other words, bitwise operators perform from the rightmost bit to the leftmost bit.

Bitwise operations

```
#!/usr/bin/perl
use warnings;
use strict;

my $a = 0b0101; # 5
my $b = 0b0011; # 3

my $c = $a & $b; # 0001 or 1
print $c, "\n";

$c = $a | $b; # 0111 or 7
print $c, "\n";

$c = $a ^ $b; # 0110 or 6
print $c, "\n";

$c = ~$a; # 11111111111111111111111111111010 (64bits computer) or 4294967290
print $c, "\n";
```

```
$c = $a >> 1; # 0101 shift right 1 bit, 010 or 2
print $c, "\n";

$c = $a << 1; # 0101 shift left 1 bit, 1010 or 10
print $c, "\n";
```

Table 1. Comparison operators for numbers

Equality	Operators
Equal	`==`
Not Equal	`!=`
Comparison	`<=>`
Less than	`<`
Greater than	`>`
Less than or equal	`<=`
Greater than or equal	`>=`

All the operators in the table above are obvious except the number comparison operator `<=>` which is also known as spaceship operator.

The number comparison operator is often used in sorting numbers. See the code below:

```
$a <=> $b
```

- **1** if `$a` is greater than `$b`
- **0** if `$a` and `$b` are equal
- **-1** if `$a` is lower than `$b`

Example

```
#!/usr/bin/perl
use warnings;
use strict;

my $a = 10;
my $b = 20;

print $a <=> $b, "\n";

$b = 10;
print $a <=> $b, "\n";

$b = 5;
print $a <=> $b, "\n";
```

5.2. String operators

Table 2. Comparison operators

Equality	Operators
Equal	`eq`
Not Equal	`ne`
Comparison	`cmp`
Less than	`lt`
Greater than	`gt`
Less than or equal	`le`
Greater than or equal	`ge`

Concatenation operators

Perl provides the concatenation (`.`) and repetition (`x`) operators that allow you to manipulate strings

Concatenation operator (`.`)

```
print "This is" . " concatenation operator" . "\n";
```

Repetition operators (x)

```
print "a message " x 4, "\n";
```

The chomp() operator

The `chomp()` operator (or function) removes the last character in a string and returns a number of characters that were removed. The `chomp()` operator is very useful when dealing with the user's input because it helps you remove the new line character \n from the string that the user entered.

```
#!/usr/bin/perl
use warnings;
use strict;

my $s;
chomp($s = <STDIN>);
print $s;
```

The `<STDIN>` is used to get input from users.

5.3. Logical operators

Logical operators are often used in control statements such as if, while, given, etc., to control the flow of the program. The following are logical operators in Perl:

- `$a && $b` performs the logic `AND` of two variables or expressions. The logical `&&` operator checks if both variables or expressions are true.
- `$a || $b` performs the logic `OR` of two variables or expressions. The logical `||` operator checks whether a variable or expression is true.
- `!$a` performs the logic `NOT` of the variable or expression. The logic `!` operator inverts the value of the following variable or expression. In the other words, it converts `true` to `false` or `false` to `true`.

6. Perl List

Perl list and how to manipulate list elements using various techniques such as list slicing, ranging and qw() function.

A Perl list is a sequence of scalar values. You use parenthesis and comma operators to construct a list. Each value is the list is called list element. List elements are indexed and ordered. You can refer to each element by its position.

6.1. Simple Perl list

```
();
(10,20,30);
("this", "is", "a","list");
```

In the example above:

- The first list `()` is an empty list.
- The second list `(10,20,30)` is a list of integers.
- The third list `("this", "is", "a","list")` is a list of strings.

Each element in the list is separated by a comma `(,)`. The print` operator is a list operator. So let's display our lists above with the `print` operator to see how it works:

```
#!/usr/bin/perl
use warnings;
use strict;

print(()); # display nothing
print("\n");
print(10,20,30); # display 102030
print("\n");
print("this", "is", "a","list"); # display: thisisalist
print("\n");
```

We passed several lists to the `print` operator to display their elements. All the lists that we have seen so far contain an element with the same data type. These lists are called simple lists.

6.2. Complex Perl list

A Perl list may contain elements that have different data types. This kind of list is called a complex list. Let's take a look at the following example:

```
#!/usr/bin/perl
use warnings;
use strict;

my $x = 10;
```

```
my $s = "a string";
print("complex list", $x , $s ,"\n");
```

6.3. Using qw function

Perl provides the `qw()` function that allows you to get a list by extracting words out of a string using the space as a delimiter. The `qw` stands for quote word. The two lists below are the same:

```
#!/usr/bin/perl
use warnings;
use strict;

print('red','green','blue'); # redgreenblue
print("\n");

print(qw(red green blue)); # redgreenblue
print("\n");
```

Similar to the `q/` and `q//` operators, you can use any non-alphanumeric character as a delimiter. The following lists are the same:

```
qw\this is a list\;
qw{this is a list};
qw[this is a list];
```

6.4. Flattening list

If you put a list, called an internal list, inside another list, Perl automatically flattens the internal list. The following lists are the same:

```
(2,3,4,(5,6))
(2,3,4,5,6)
((2,3,4),5,6)
```

6.5. Accessing list element

You can access elements of a list by using the zero-based index. To access the n^{th} element, you put (n – 1) index inside square brackets.

Let's take a look at the following example:

```
#!/usr/bin/perl
use warnings;
use strict;

print(
     (1,2,3)[0] # 1 first element
```

```
);
print "\n"; # new line

print(
     (1,2,3)[2] # 3 third element
);
print "\n"; # new line
```

To get multiple elements of a list at a time, you can put a list inside square brackets. This feature is called list slice. You can omit the parenthesis of the list inside the square bracket.

```
(1,2,3,4,5)[0,2,3] # (1,3,4)
```

The above code returns a list of three elements `(1, 3, 4)`.

6.6. Ranges

Perl allows you to build a list based on a range of numbers or characters e.g., a list of numbers from 1 to 100, a list of characters from a to z. The following example defines two lists:

```
(1..100)
(a..z)
```

7. Array

Perl array and how to use arrays effectively in your program and techniques to manipulate array's elements.

A list is immutable so you cannot change it directly. In order to change a list, you need to store it in an array variable.

By definition, an array is a variable that provides dynamic storage for a list.

In Perl, the terms array and list are used interchangeably, but you have to note an important difference: a list is immutable whereas an array is mutable. In other words, you can modify the array's elements, grow or shrink the array, but not a list.

A scalar variable begins with the dollar sign (`$`), however, an array variable begins with an at-sign (`@`).

How to declare an array variable

```
#!/usr/bin/perl
use warnings;
use strict;

my @days = qw(Mon Tue Wed Thu Fri Sat Sun);
print("@days", "\n");
```

The `$` sign looks like `S` in the word scalar. And `@` looks like `a` in the word array, which is a simple trick to remember what type of variables you are working with.

7.1. Accessing Perl array elements

Like a list, you can access array elements using square brackets `[]` and indices.

Example

```
#!/usr/bin/perl
use warnings;
use strict;

my @days = qw(Mon Tue Wed Thu Fri Sat Sun);
print($days[0]);

print("\n");
```

If you take a look at the code carefully, you will see that we used `$days[0]` instead of `@days[0]`.

This is because an array element is a scalar, you have to use the scalar prefix (`$`). In Perl, the rule is that the prefix represents what you want to get, not what you've got.

Perl also allows you to access array elements using negative indices. Perl returns an element referred to by a negative index from the end of the array. For example, `$days[-1]` returns the last

element of the array `@days`.

You can access multiple array elements at a time using the same technique as the list slice.

```
#!/usr/bin/perl
use warnings;
use strict;

my @days = qw(Mon Tue Wed Thu Fri Sat Sun);
my @weekend = @days[-2..-1]; # SatSun

print("@weekend");
print("\n");
```

7.2. Counting Perl array elements

If you treat an array as a scalar, you will get the number of elements in the array.

```
my $count = @days;
```

This code causes an error in case you don't really want to count it but accidentally assign an array to a scalar. To be safe, use the `scalar()` function.

```
my @days = qw(Mon Tue Wed Thu Fri Sat Sun);
my $count = scalar @days;
print($count, "\n");
```

The operator `$#` returns the highest index of an array.

```
my @days = qw(Mon Tue Wed Thu Fri Sat Sun);
my $last = $#days;
print($last, "\n"); #" 6
```

7.3. Modifying Perl array elements

To change the value of an element, you access the element using the index and assign it a new value. Perl also allows you to change the values of multiple elements at a time.

Example

```
#!/usr/bin/perl
use warnings;
use strict;

my @days = qw(Mon Tue Wed Thu Fri Sat Sun);

$days[0] = 'Monday';
```

```
@days[1..6] = qw(Tuesday Wednesday Thursday Friday Saturday Sunday);

print("@days","\n");
```

7.4. Perl array operations

Perl provides several useful functions and operators to help you manipulate arrays effectively. We will cover the most important ones in the following sections.

Perl array as a stack with push() and pop() functions

Both functions treat an array as a stack. A stack works based on the last in first out (LIFO) philosophy. It works exactly the same as a stack of books. The `push()` function appends one or more elements to the end of the array, while the `pop()` function removes the last element from the end of the array.

How to use `push()` and `pop()` functions

```
#!/usr/bin/perl
use warnings;
use strict;

my @stack = (); # empty array

print("push 1 to array\n");

push(@stack,1);

print("push 2 to array\n");
push(@stack,2);

print("push 3 to array\n");
push(@stack,3);

print("@stack", "\n");

my $elem = pop(@stack);
print("element: $elem\n");

$elem = pop(@stack);
print("element: $elem\n");

$elem = pop(@stack);
print("element: $elem\n");
```

Perl array as a queue with `unshift()` and `pop()` functions

If the `push()` and pop()` treat an array as a stack, the `unshift()` and `pop()` functions treat an array as a queue. A queue works based on the first in first out (FIFO) philosophy. It works like a queue of visitors. The `unshift()` function adds one or more elements to the front of the array, while the `pop()` function removes the last element of the array.

How to use `unshift()` and `pop()` functions

```
#!/usr/bin/perl
use warnings;
use strict;

my @queue = (); # empty queue

print("enqueue 1 to array\n");
unshift(@queue,1);

print("enqueue 2 to array\n");
unshift(@queue,2);

printf("enqueue 3 to array\n");
unshift(@queue,3);

print("@queue", "\n"); # 3 2 1

my $elem = pop(@queue);
print("element: $elem\n");

$elem = pop(@queue);
print("element: $elem\n");

$elem = pop(@queue);
print("element: $elem\n");
```

Sorting Perl arrays

Perl provides the `sort()` function that allows you to sort an array in alphabetical or numerical order.

Sorting an array of strings alphabetically

```
#!/usr/bin/perl
use warnings;
use strict;

my @fruits = qw(oranges apples mango cucumber);
my @sorted = sort @fruits;

print("@sorted","\n"); # apples cucumber mango oranges
```

The `sort()` function also accepts a block of code that allows you to change the sort algorithm. If you want to sort an array in numerical order, you need to change the default sorting algorithm.

```
#!/usr/bin/perl
use warnings;
use strict;
my @a = qw(3 2 1 4 7 6);
print("unsorted: ", "@a", "\n"); # unsorted: 3 2 1 4 7 6
```

```
@a = sort {$a <=> $b} @a;
print("sorted:", "@a", "\n"); # sorted: 1 2 3 4 6 7
```

In the example above:

- First, we had an unsorted array `@a`, and we displayed the `@a` array to make sure that it is unsorted.
- Second, we used the `sort()` function to sort the `@a` array. We passed a block of code `{$a <=> $b}` and the `@a` array to the sort function. The `$a` and `$b` are global variables defined by the `sort()` function for sorting. The operator `<=>` is used to compare two numbers. The code block `{$a <=> $b}` returns -1 if `$a` < `$b`, 0 if `$a` = `$b`, and 1 if `$a` > `$b`.
- Third, we displayed the elements of the sorted array @a.

For more information on the `sort()` function, check out the Perl sort function Perl sort function.

8. Hash

Another compound data type called Perl hash and how to manipulate hash elements effectively.

A Perl hash is defined by key-value pairs. Perl stores elements of a hash in such an optimal way that you can look up its values based on keys very fast.

With the array, you use indices to access its elements. However, you must use descriptive keys to access hash element. A hash is sometimes referred to as an associative array.

Like a scalar or an array variable, a hash variable has its own prefix. A hash variable must begin with a percent sign (%). The prefix `%` looks like key/value pair so remember this trick to name the hash variables.

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

Perl provides the `=>` operator as an alternative to a comma (,). It helps differentiate between keys and values and makes the code more elegant.

When you see the `=>` operator, you know that you are dealing with a hash, not a list or an array.

```
my %countries =  ( England => 'English',
                   France  => 'French',
                   Spain   => 'Spanish',
                   China   => 'Chinese',
                   Germany => 'German');
```

Perl requires the keys of a hash to be strings, meanwhile, the values can be any scalars. If you use non-string values as the keys, you may get an unexpected result.

In addition, a hash key must be unique. If you try to add a new key-value pair with the key that already exists, **the value of the existing key will be over-written**.

You can omit the quotation in the keys of the hash.

8.1. Hash Operations

The most commonly used operation in the hash.

Look up Perl hash values

Use a hash key inside curly brackets {} to look up a hash value.

```
#!/usr/bin/perl
use warnings;
use strict;
# defines country => language hash
my %langs = ( England => 'English',
              France => 'French',
              Spain => 'Spanish',
```

```
              China => 'Chinese',
              Germany => 'German');
# get language of England
my $lang = $langs{'England'}; # English
print($lang,"\n");
```

Add new element

```
$langs{'Italy'} = 'Italian';
```

Remove a single key/value pair

```
delete $langs{'China'};
```

Modify hash elements

```
# add new key value pair
$langs{'India'} = 'Many languages';
# modify official language of India
$langs{'India'} = 'Hindi'; #
```

Loop over Perl hash elements

Perl provides the `keys()` function that allows you to get a list of keys in scalars. You can use the `keys()` function in a `for` loop statement to iterate the hash elements:

```
#!/usr/bin/perl
use warnings;
use strict;

# defines country => language hash
my %langs = ( England => 'English',
              France  => 'French',
              Spain   => 'Spanish',
              China   => 'Chinese',
              Germany => 'German');

for (keys %langs) {
    print("Official Language of $_ is $langs{$_}\n");
}
```

The `keys()` function returns a list of hash keys. The `for` loop visits each key and assigns it to a special variable `$_`. Inside the loop, we access the value of a hash element via its key as `$langs{$_}`.

9. *if* Statement

Perl `if` statement allows you to control the execution of your code based on conditions. The simplest form of the `if` statement is as follows:

```
if (expression);
```

In this form, you can put the `if` statement after another statement. Let's take a look at the following example:

```
my $a = 1;
print("Welcome to Perl if tutorial\n") if ($a == 1);
```

The message is only displayed if the expression `$a == 1` evaluates to `true`.

How Perl defines *true* and *false*?

- Both number 0 and string "0" are `false`.
- The `undefined` value is `false`.
- The empty list `()` is `false`.
- The empty string `""` is `false`.
- Everything else is `true`.

Execute multiple statements based on a condition

```
if (expression) {
   statement;
   statement;
   ...
}
```

The curly braces `{}` are required even if you have a single statement to execute

if..else Statement

Perl provides the `if else` statement that allows you to execute a code block if the expression evaluates to `true`, otherwise, the code block inside the `else` branch will execute.

```
if (expression) {
   // if code block;
} else {
   // else code block;
}
```

Example

```
my $a = 1;
my $b = 2;
if ($a == $b) {
    print("a and b are equal\n");
} else {
    print("a and b are not equal\n");
}
```

The code block in the else branch will execute because $a and $b are not equal.

if..elsif statement

In some cases, you want to test more than one condition:

- If `$a` and `$b` are equal, then do this.
- If `$a` is greater than `$b` then do that.
- Otherwise, do something else.

Perl provides the if elsif statement for checking multiple conditions and executing the corresponding code block:

```
if (expression) {
    ...
} elsif (expression2) {
    ...
} elsif (expression3) {
    ...
} else {
    ...
}
```

10. A Simple Program

We are going to apply what we have learned so far to create a simple program called currency converter.

- We will use a hash to store the exchange rates.
- To get the inputs from users via the command line, we will use `<STDIN>`. We use the `chomp()` function to remove the newline character (\n) from the user's inputs.
- We convert the amount from local currency to foreign currency if the currencies are supported.

```
#!/usr/bin/perl
use warnings;
use strict;

my ($l_curr, $f_curr, $l_amount, $f_amount);

my %rates = (
     USD => 1,
     YPY => 82.25,
     EUR => 0.78,
     GBP => 0.62,
     CNY => 6.23
);

# print supported currencies
print("Supported currency:\n");
for (keys %rates) {
    print(uc($_),"\n");
}

# get inputs from users:
print("Enter local currency:\n");
$l_curr = <STDIN>;

print("Enter foreign currency:\n");
$f_curr = <STDIN>;

print("Enter amount:\n");
$l_amount = <STDIN>;

chomp($l_curr,$f_curr,$l_amount);

# check user's inputs.
if (not exists $rates{$l_curr}) {
     print("Local currency is not supported\n");
} elsif (not exists $rates{$f_curr}) {
     print("Foreign currency is not supported\n");
} else {
     # convert from local currency to foreign currency
     $f_amount = ($rates{$f_curr} / $rates{$l_curr}) * $l_amount;
```

```
    # print out the result
    print("$l_amount $l_curr = $f_amount $f_curr","\n");
}
```

11. *unless* Statement

Before discussing the unless statement, let's revisit Perl's philosophy:

> There is more than one way to do it.

Perl always provides you with an alternative way to achieve what you need to do. In programming, you often hear something like this:

- If it's not true, then do this (use `if` not statement).
- or unless it's true, then do this (use `unless` statement).

The effect is the same but the philosophy is different. That's why Perl invented the `unless` statement to increase the readability of code when you use it properly.

the Perl `unless` statement is equivalent to the `if not` statement

unless statement

```
statement unless (condition);
```

Perl executes the statement from right to left, if the condition is `false`, Perl executes the statement that precedes the `unless`. If the condition is `true`, Perl skips the statement.

If you have more than one statement to execute, you can use the following form of the Perl unless statement:

```
unless (condition) {
    // code block
}
```

If the `condition` evaluates to `false`, Perl executes the code block, otherwise, it skips the code block.

```
my $a = 10;

unless($a <= 0){
    print("a is greater than 0\n")
}
```

unless..else statement

Sometimes you want to say unless the condition is `true`, then do this, otherwise do that.

This is where the `unless…else` statement comes into play. See the following `unless else` statement:

```
unless (condition) {
```

```
    // unless code block
} else {
    // else code block
}
```

If the condition is `false`, Perl will execute the `unless` code block, otherwise, Perl will execute the `else` code block.

```
my $a = 10;
unless ($a >= 0) {
    print("a is less than 0\n");
} else {
    print("a is greater than or equal 0\n");
}
```

```
a is greater than or equal 0
```

unless..elsif..else statement

If you have more than one condition for checking with the `unless` statement, you can use the `unless elsif else` statement as follows:

```
unless (condition_1) {
    // unless code block
} elsif (condition_2) {
    // elsif code block
} else {
    // else code block
}
```

You can have many `elsif` clauses in the `unless elsif` statement.

```
my $a = 1;

unless ($a > 0) {
    print("a is less than 0\n");
} elsif ($a == 0) {
    print("a is 0\n");
} else {
    print("a is greater than 0\n");
}
```

unless statement guidelines

You should use the `unless` statement with a simple condition to improve the code readability, especially when used as a postfix after another statement like the following example:

```
my $a = 1;
print("unless used with a very simple condition ONLY.\n") unless($a <
0);
```

You should avoid using the `unless` statement when the condition is complex and requires `else` and/or `elsif` clauses.

If you take a look at the following code, it is difficult to interpret the meaning of the condition.

```
my $a = 1;
my $b = 10;
my $c = 20;
unless ($a < 0 && $b == 10 && $c > 0) {
    print("unless used with a very complex condition\n");
}
```

12. *given* Statement

Perl `given` statement, is similar to the switch case statement in other languages.

The `given` statement works like a series of if statements that allow you to match an expression or variable against different values, depending on the matched value, Perl will execute statements in the corresponding `when` clause.

Pragma for using given statement

Perl introduced the `given` statement since version 5.10. In order to use the Perl `given` statement, you must use the following pragma:

```
use v5.10;
```

Or use the following pragma:

```
use feature "switch";
```

Perl *given* Syntax

There are several new keywords introduced along with the `given` such as: `when`, `break` and `continue`.

```
given (expr) {
     when (expr1) { statement; }
     when (expr1) { statement; }
     when (expr1) { statement; }
     ...
}
```

given statement in greater detail

- Both given and `when` accept arguments in a scalar context.
- The type of argument you pass to the `given` clause determines the kind of pattern matching that Perl will use to perform matching. If the argument appears to be a `Boolean` expression, Perl evaluates it directly. Otherwise, Perl will use the smart match operator to evaluate the argument, something like `$_ ~~ expr`
- To break out a `when` block, you use the `break` statement. Perl uses `break` statement implicitly for all `when` blocks so you don't have to explicitly specify it.
- To fall through from one case to the next, you use the `continue` statement.

- From version 5.12 you can use `when` as a statement modifier.
- From version 5.14, the `given` statement returns the last evaluated expression if no condition is `true` or the last evaluated expression of the `default` clause. The `given` statement also returns an empty list when the

break statement is encountered or no condition is matched.

given statement examples

The following program asks the user to input an RGB (red, green, blue) color and returns its color code:

```
use v5.10; # at least for Perl 5.10
#use feature "switch";
use warnings;
use strict;

my $color;
my $code;

print("Please enter a RGB color to get its code:\n");
$color = <STDIN>;
chomp($color);
$color = uc($color);

given($color){
    when ('RED') {  $code = '#FF0000'; }
    when ('GREEN') {  $code = '#00FF00'; }
    when ('BLUE') {  $code = '#0000FF'; }
    default {
        $code = '';
    }
}
if ($code ne '') {
    print("code of $color is $code \n");
} else {
    print("$color is not RGB color\n");
}
```

How program works

- First, we declared the `pragma use v5.10;` in order to use the `given` statement.
- Second, we asked the user for a color, we removed the newline by using the `chomp()` function and made the input color upper case so that whatever format of color the user entered is accepted e.g., Red, rEd or RED is the red color.
- Third, we used the `given` statement to check. If no color is found, then we set the color code to blank in the default clause. Based on the user's input, we got the corresponding color code and display it.

From Perl version 5.12, you can use the `when` statement as a statement modifier like the following example:

```
given ($color) {
    $code = '#FF0000' when 'RED';
    $code = '#00FF00' when 'GREEN';
```

```
    $code = '#0000FF' when 'BLUE';
    default { $code = ''; }
}
```

In addition, the given statement returns a value that is the result of the last expression.

```
print do {
    given ($color) {
        "#FF0000\n" when 'RED';
        "#00FF00\n" when 'GREEN';
        "#0000FF\n" when 'BLUE';
        default { ''; }
    }
}
```

More complex example:

```
use v5.12;
use strict;
use warnings;

print 'Enter something: ';
chomp( my $input = <> );

print do {
    given ($input) {
        "The input has numbers\n"  when /\d/;
        "The input has letters\n"  when /[a-zA-Z]/;
        default { "The input has neither number nor letter\n"; }
    }
}
```

How the program works

- The program asks the user to enter anything that can be numbers, letters, or both.
- In the `when` clause, we used a very special expression that is known as a regular expression. The `/\d/` matches any string that contains only numbers, The `/[a-zA-Z]/` expression matches the string that contains only letters. In this case, the `given` statement can do more advanced matches.

13. *for* Loop

Perl for loop statement to loop over elements of a list.

for and *foreach* statements

The Perl `for` loop statement allows you to loop over elements of a list. In each iteration, you can process each element of the list separately. This is why the `for` loop statement is sometimes referred to as `foreach` loop.

In Perl, the `for` and `foreach` loop are interchangeable, therefore, you can use the `foreach` keyword in where you use the `for` keyword.

Loop over elements of an array

```
#!/usr/bin/perl
use warnings;
use strict;

my @a = (1..9);
for (@a) {
    print("$_","\n");
}
```

How it works

- First, we defined an array of 9 integers `@a`
- Second, we used `for` loop statement to loop over elements of the `@a` array.
- Third, inside the loop, we displayed element's value using default variable `$_`

If you replace the `for` keyword by the `foreach` keyword in the above example, it works the same.

```
#!/usr/bin/perl
use warnings;
use strict;

my @a = (1..9);
foreach (@a) {
    print("$_","\n");
}
```

for loop iterator

If we don't supply an explicit iterator to the loop, Perl will use a special variable called default variable with the name `$_` as the iterator. In each iteration, Perl assigns each element of the array `@a` to the default variable `$_`.

Explicit Perl for loop iterator

If you want to specify an explicit iterator for the loop, you can declare it in the `for` loop statement as follows:

```
#!/usr/bin/perl
use warnings;
use strict;

my @a = (1..9);
for my $i (@a) {
    print("$i","\n");
}
```

`$i` is the iterator of the `for` loop in this example. In each iteration, Perl assigns the corresponding element of the array to the `$i` iterator. Notice that the `$i` variable exists only during the execution of the loop.

If you declare an iterator before entering the loop, Perl will restore its original value after the loop is terminated. Take a look at the following example:

```
#!/usr/bin/perl
use warnings;
use strict;

my @a = (1..9);
my $i = 20;

for $i (@a) {
    print("$i","\n");
}

print('iterator $i is ',"$i","\n"); # 20
```

How it works

- First, we declared variable `$i` before the loop and initialized its value to `20`.
- Second, we used a variable `$i` as the iterator; its value changes in each iteration of the loop.
- Third, after the loop, we displayed the value of `$i`. Perl restored its original value, which is `20`.

Perl for loop iterator: value or alias

In each iteration of the loop, Perl creates an alias instead of a value. In other words, if you make any changes to the iterator, the changes also reflect in the elements of the array. See the following example:

```
#!/usr/bin/perl
use warnings;
use strict;

my @b = (1..5);

print("Before the loop: @b \n");

for (@b) {
    $_ = $_ * 2;
```

```
}

print("After the loop: @b \n");
```

How it works

- First, we declared an array `@b` with 5 elements from 1 to 5. We displayed the array `@b` elements using `print` function
- Second, we iterated elements of the array. We multiplied each element with `2` through the iterator `$_`
- Third, outside of the loop, we displayed the elements of the array again

C-style *for* loop

Perl also supports for loop in C-style. However, it is not a good practice to use the C-style for loop because to code will become less readable.

```
for (initialization; test; step) {
    // code block;
}
```

There are three control parts:

- Initialization. Perl executes the initialization once when the loop is entered. We often use initialization to initialize a loop counter variable.
- Test. Perl evaluates the `test` expression at the beginning of each iteration and executes the code block inside the loop body as long as the test expression evaluates to false.
- Step. Perl executes `step` at the end of each iteration. You often use the step to modify the loop counter.

C-style for loop:

```
#!/usr/bin/perl
use warnings;
use strict;

my @c = (1..6);
for (my $i = 0; $i <= $#c; $i++) {
    print("$c[$i] \n");
}
```

It is much more readable if you Perl's for loop style

```
#!/usr/bin/perl
use warnings;
use strict;

my @c = (1..6);
for (@c) {
    print("$_ \n");
```

```
}
```

14. *while* loop statement

Perl `while` loop statement executes a code block repeatedly as long as the test condition remains `true`. The test condition is checked at the beginning of each iteration.

```
while (condition) {
    # code block
}
```

If the `condition` evaluates to `true`, the code block inside `while` loop executes.

At the beginning of each iteration, the `condition` is reevaluated. The loop is terminated if the `condition` evaluates to `false`.

At some point in the loop, you have to change some variables that make the condition `false` to stop the loop. Otherwise, you will have an indefinite loop that makes your program execute until the stack overflow error occurs.

The `while` loop statement has an optional block: `continue`, which executes after each current iteration. In practice, the `continue` block is rarely used.

If you want to execute a code block as long as the condition is `false`, you can use `until` statement.

In case you want to check the condition at the end of each iteration, you use the `do…while` or `do…until` statement instead.

To control the loop, you use the `next` and `last` statements.

Example

Happy New Year Count Down program

```
#!/usr/bin/env perl
use warnings;
use strict;

my $counter = 10;

while ($counter > 0) {
    print("$counter\n");
    $counter--; # count down
    sleep(1);   # pause program for 1 second
    if ($counter == 0) {
        print("Happy New Year!\n");
    }
}
```

Let's examine the code above in more detail

- First, declare a `$counter` variable and set its value to `10`.
- Next, put a condition to make sure that the value of `$counter` is greater than zero before entering into the loop.
- Then, displayed the `$counter` and decreased its current value of one. We used the `sleep()` function to pause the program for a second in each iteration.
- After that, use the *if statement* to check if `$counter` is zero to print the "Happy New Year" message. The code block inside the loop executes 10 times before the `$counter` is set to zero.
- Finally, after each iteration, the `$counter` decreases, and its value is set to zero at the 10th iteration. Perl terminated the loop.

while loop with diamond operator `<>`

You often use the while loop statement with the diamond operator `<>` to get the user's input from the command line:

```
#!/usr/bin/perl
use warnings;
use strict;

my $num;
my @numbers = ();

print "Enter numbers, each per line :\n";
print "ctrl-z (windows) or ctrl-d(Linux) to exit\n>";

while(my $input = <>) {
    print(">");
    chomp $input;
    $num = int($input);
    push(@numbers, $num);
}

print "You entered: @numbers\n";
```

How it works

- First, assign the user's input to the `$input` variable using the diamond operator (`<>`). Because it doesn't specify any filehandle for the diamond operator, Perl checks the special array `@ARGV`, which is empty in this case, hence instructs the diamond operator to read from `STDIN` i.e., from the keyboard.
- Second, remove the newline character from the `$input` variable using the `chomp()` function and convert `$input` to an integer.
- Third, add the integer into the `@number` array.

while loop statement modifier

let's take a look at the following example:

```
#!/usr/bin/perl
use warnings;
use strict;

my $i = 5;
print($i--,"\n") while ($i > 0);
```

The `while` loop statement is placed after another statement.

Perl evaluates the statements from right to left.

It means that Perl evaluates the condition in the `while` statement at the beginning of each iteration.

You use the `while` loop statement modifier only if you have one statement to execute repeatedly based on a condition like the above example.