

17. Perl

Scalar Types

Scalar Literals

Scalar literals are single values of one of three kinds:
numbers, character strings, or references.

In Perl, unlike most languages, most numeric data is represented in a single data type: double-precision floating point.

There are two kinds of string literals:

- those delimited by single quotes (`'`) and
- those delimited by double quotes (`"`).

Double-quoted string literals differ from single-quoted ones in that

1. They can include escape sequences (e.g., `\t`, `\n`) to encode special characters.
2. Variable names that appear in them are interpolated (converted to their values).

Scalar Variables

The name of a scalar variable begins with `$`, followed by a letter, and possibly followed by a string of letters, digits, and underscores.

Perl is case sensitive.

By convention, user-defined variables do not use uppercase letters.

Underscores are preferred over embedded uppercase letters.

So `$array_sum` is preferred over `$arraySum`.

Scalar variables aren't declared to have specific types and often aren't explicitly declared.

A scalar variable can store a number, a string, or a reference.

The appearance of a scalar variable causes the compiler to implicitly declare it.

All implicitly declared variables are global.

We discuss scalar variable declarations when they become relevant.

Perl includes a large number of implicit variables, many of them scalar.

To illustrate variable and escape-sequence interpolation, suppose the value of `$sum` is 8.

Then

```
"The sum is:\t $sum"
```

is interpreted as

```
"The sum is: 8"
```

Scalar Operators

Perl has all the arithmetic operators of C/C++, including the increment and decrement operators.

Perl also has a right-associative exponentiation operator, `**`.

Its precedence is higher than `*` and `/` and lower than unary `-`.

So `2 ** 3 ** 2 * 3` is $2^9 * 3$ (= 1536).

String Operators

Strings are not stored as arrays of characters.

Individual characters and substrings are accessed only through predefined functions.

The dot operator (`.`) is the concatenation operator.

The repetition operator, `x`, replicates its left, string operand the number of times specified by its right, numerical operand.

For example,

```
"ha " x 3
creates the string
"ha ha ha "
```

The right operand of `x` can be any arithmetic expression.

Its value is coerced to an integer.

`x` has higher precedence than dot (`.`).

```
So
"Here " . "now " x 2
is
"Here now now "
```

String operators coerce their values to be strings.

```
So
1 . 2 x 3
is
"1222"
```

When a scalar variable is used as a string but has the value `undef`, it's interpreted as the empty string, `" "`.

String Functions

Functions in Perl

Perl functions are often used for their side effects – the changes made to their arguments – and not for their returned values.

When the returned value is of no interest, the function call can appear by itself:

```
fun( x )
```

A Perl function can often be treated as either an operator or a function.

If you think of it as a function, you delimit the actual parameters in a call to it with parentheses.

If you think of it as an operator, you list its operands after the operator.

Suppose `fun` is a function that uses `$_` as its default actual parameter (or operand).

I.e., `fun` uses `$_` as its parameter value when invoked by a call that doesn't include an actual parameter.

Such a call can be specified with either of the following forms:

```
fun( )
```

```
fun
```

Predefined String Functions

`chop` is used for its side effect: removing the last character from its argument (which should not be a literal).

For example, if the value of `$pets` is "cats", then

```
chop( $pets )
```

changes the value of `$pets` to "cat".

`chop` returns the character removed ("s" in the last example).

`chop` used to be used to remove the newline character from an input line.

But, while UNIX represents newline with a single character (a carriage return), some other systems represent it with two characters (carriage return-line feed).

So it's better to consider *input record separators* rather than newlines.

Sometimes the input record separator is one character, sometimes two.

`chomp` removes whatever the system uses for an input record separator.

It returns the number of input record separators removed.

If `chomp` is used on a string not ending in an input line separator, it does nothing to the string.

Implicit variable `$\` is initially set to the character(s) used for the input record separator.

You can set `$\` to whatever you want and use `chomp` to remove that from the given string.

Both `chop` and `chomp` use the implicit variable `$_` as their default parameter.

Both `chop` and `chomp` can take a list of strings as parameters.

Then `chop` removes the last character of each parameter and returns the character removed from the first parameter.

`chomp` removes the input record separator from each parameter that has one and returns the number removed.

For example, if the values of `$a`, `$b`, and `$c` are "a", "to", and "dog", respectively, then

```
chop( $a, $b, $c )
```

changes `$a` to "", `$b` to "t", and `$c` to "do" and returns "a".

`chomp` isn't needed to remove input record separators from numeric input.

A string in an expression where a number is expected is converted to a number.

This eliminates any input record separator.

`length` takes a string argument and returns the number of characters in it.

An input record separator counts as one character even when it actually consists of two characters.

For example, if the value of `$str` is "cars",

```
length( $str )
```

returns 4.

`lc` and `uc` each take a string argument.

`lc` converts any uppercase letter in the string to lowercase.

`uc` converts any lowercase letter in the string to uppercase.

`index` takes two string arguments and searches the first from left to right for the first occurrence of the second.

It returns the index in the first where the second is found, -1 if the second isn't found.

The index of the leftmost character is 0.

`rindex` does the same, but searches from right to left.

For example,

```
index( "binding", "in" ) returns 1
```

```
rindex( "binding", "in" ) returns 4
```

`substr` is used to extract substrings. It's used as

```
substr( string, position, length )
```

which returns the substring of *string* beginning at index *position* and of length *length*.

If *position* < 0, the position is counted from the right.

If *length* is omitted, all characters from *position* to the right end of *string* are returned.

For example, if the value of `$str` is "animal",

```
substr( $str, 1, 2 ) returns "ni"
```

```
substr( $str, -3, 2 ) returns "ma"
```

```
substr( $str, 3 ) returns "mal"
```

`join` is used to build a new string from two or more existing strings.

It's used in the form

```
join separator, list
```

where

- *list* is the list of strings to be joined together and
- *separator* separates these parts of the new string.

For example,

```
join ':' , $month, $day, $year
```

`split` takes a string apart.

We discuss it when we come to arrays.

Mixed-Mode Expressions

When a numeric operand occurs with a string operator, it's coerced to a string.

When a string operand occurs with a numeric operator, it's coerced to a number.

Leading white space and trailing nondigit characters are ignored.

If the string doesn't then include a number, the value is 0.

For example, if `$str` is " 21nm", then

`5 + $str` converts `$str` to 21, giving 26.

`5 . $str` converts 5 to "5", giving "5 21nm".

`- $str` converts `$str` to 21, giving -21.

`5 + "a7"` convertst "a7" to 0, giving 5.

Perl provides no function or operator for explicitly converting the type of a value.

But the only case where the unary plus operator has any effect is when the operand is a string.

Assignment Statements

Assignment, =, is as in C/C++.

Every Perl statement must be terminated by a semicolon unless it's the last statement in a block (discussed later).

As in C/C++, the assignment operator produces a result, namely, the value assigned.

The precedence of = is lower than that of all the operators discussed so far.

For example, if \$num is 3, then

```
$x = $y = $num + 2;
```

assigns 5 to \$y then to \$x.

The result of an assignment can be taken as either

- the assigned value (an *r-value*) or
- the address of the target variable (an *l-value*).

For example, the following chops the new value of \$str:

```
chomp( $str = $str1 . $str2 )
```

As in C/C++, an expression is turned into a statement by following it with a semicolon – e.g.,

```
$a++;
```

Perl, like C/C++, has compound assignment operators – e.g.,

```
$a += 3;
```

Simple Input/Output

Files are specified with special variables called *filehandles*.

The preassigned filehandle for standard input (the keyboard) is `STDIN`.

The preassigned filehandle for standard output (the screen) is `STDOUT`.

To perform input, the filehandle is placed within `<...>`.

The angled brackets form the operator, called the *line input* (or *angle*) *operator*.

The expression `<STDIN>` reads keyboard input up to, and including, an input record separator.

Often `<STDIN>` is on the right side of an assignment to a scalar variable – e.g.,

```
$new_line = <STDIN>;
```

When the input is used as a string, we often don't want the input record separator, so we use

```
chomp( $new_line = <STDIN> );
```

When the end of file is found, `<STDIN>` returns `undef`.

The keyboard input specifying end of file is system dependent.

In UNIX, it's Control-D.

In Windows, it's Control-Z.

For output, `print` takes a filehandle and any number of string parameters to output.

`STDOUT` is the default file handle.

`print` can be considered either a function or an operator.

Examples

```
print( "The ", "cat\n" );  
print "The sum is: $sum",  
      "\tThe average is: $average\n";
```

The `printf` function gives more control over the output format.

Its first parameter is a string literal specifying the form of the output.

It includes format codes for the values of variables.

The remaining parameters are the names of the variables whose values are to be output.

Where n is an integer specifying the width of the field were the value appears, the format codes are

`%ns` for strings

`%nd` for integers

`%n.mf` for floating point values – where m is the number of digits to display to the right of the decimal point

All characters in the format code parameter that aren't part of the format codes are copied as-is to the output.

Example

```
$i = 16;
```

```
$x = 21.5;
```

```
$day = "Wednesday";
```

```
printf
```

```
    "Today is %9s.\t\t$i = %4d and \ $x = %6.2f.\n",
    $day, $i, $x;
```

This outputs

```
Today is Wednesday.      $i = 16 and $x = 21.50.
```

The `sprintf` function is identical to `printf` but returns its (string) output rather than outputting it.

It's useful for converting numbers to strings.

For example, if `$x` is 21.68, then

```
$str_x = sprintf( "%5.1f", $x );
```

sets `$str_x` to " 21.7".

An Example

The following is a Perl program that

- prompts for the radius of a circle,
- inputs and `chomp`s the radius,
- computes the area and circumference of the circle, then
- outputs the information on the circle.

A comment is introduced with a `#` and extends to the end of the line.

The extension for Perl program files is `.pl`.

```
# Program circle.pl
$pi = 3.14159;
print "Enter the circle radius: ";
chomp($radius = <STDIN>);
$area = $pi * $radius * $radius;
$circumference = 2 * $pi * $radius;
print "A circle of radius $radius",
      " has an area of $area \n",
      " and a circumference of $circumference \n";
```

Running Perl Programs

One way to run a Perl program is

- to execute the `perl` command in a DOS window and
- include the Perl program file name as an argument.

For example, we can execute the `circle.pl` program in a DOS window with

```
C:\someDirectory> perl circle.pl
```

Single-statement programs can be run by typing the `-e` flag and the statement, delimited with double quotes, on the `perl` command line.

Example

```
C:\someDirectory> perl -e "print 'Hello\n';"
```

(UNIX implementations require the statement to be delimited by single quotes.)

You can also type an entire program as input directly to the Perl system.

Type `perl` on the command line, then as many lines of Perl code as you like, then Control-Z (Control-D in UNIX).

Example

```
C:\someDirectory> perl  
print "Hello world\n";  
print "Good-bye\n";  
Control-Z
```

Perl programs are compiled into an intermediate form, which is executed.

The compilation phase finds syntax errors.

To avoid the execution phase, use the `-c` flag.

The `-w` flag causes the Perl compiler to issue warnings when it finds suspicious (yet syntactically correct) code.

It is a good idea always to use this flag.

Example

Adding the line

```
print 3 + "cat";
```

to the end of the `circle.pl` program and executing it with the `-w` flag in a DOS window gives

```
C:\User\PerlPractice>perl -w circle.pl
Argument "cat" isn't numeric in addition (+) at circle.pl line 10.
Enter the circle radius: 2
A circle of radius 2 has an area of 12.56636
and a circumference of 12.56636
3
C:\User\PerlPractice>
```

The warnings are sometimes mixed in with the program output.