

Perl

From Wikipedia, the free encyclopedia

This is the current revision of Perl (<http://en.wikipedia.org/wiki/Perl>) as edited by Galoubet (Talk | contribs) at 12:22, 29 January 2009. This URL is a permanent link to this version of this page.

(diff) ← Previous revision | Current revision (diff) | Newer revision → (diff)

In computer programming, **Perl** is a high-level, general-purpose, interpreted, dynamic programming language. Perl was originally developed by Larry Wall, a linguist working as a systems administrator for NASA, in 1987, as a general purpose Unix scripting language to make report processing easier.^{[1][2]} Since then, it has undergone many changes and revisions and become widely popular among programmers. Larry Wall continues to oversee development of the core language, and its coming version, Perl 6.

Perl borrows features from other programming languages including C, shell scripting (sh), AWK, and sed.^[3] The language provides powerful text processing facilities without the arbitrary data length limits of many contemporary Unix tools,^[4] facilitating easy manipulation of text files. It is also used for graphics programming, system administration, network programming, applications that require database access and CGI programming on the Web. Perl is nicknamed "the Swiss Army chainsaw of programming languages" due to its flexibility and adaptability.^[5]

Contents

- 1 History
 - 1.1 Name
 - 1.2 The camel symbol
- 2 Overview
 - 2.1 Features
 - 2.2 Design
 - 2.3 Applications
 - 2.4 Implementation
 - 2.5 Availability
 - 2.5.1 Windows
- 3 Language structure
 - 3.1 Data types
 - 3.2 Control structures
 - 3.3 Subroutines
 - 3.4 Regular expressions
 - 3.4.1 Uses
 - 3.4.2 Syntax
- 4 Database interfaces
- 5 Comparative performance
 - 5.1 Optimizing
- 6 Future
- 7 The Perl community
 - 7.1 State of the Onion
 - 7.2 Pastimes
 - 7.2.1 JAPHs
 - 7.2.2 Perl golf

Perl



Paradigm	multi-paradigm: functional, imperative, object-oriented (class- based)
Appeared in	1987
Designed by	Larry Wall
Latest release	5.10.0/ 18 December 2007
Typing discipline	Dynamic
Influenced by	AWK, Smalltalk 80, LISP, C, C++, Pascal, sed, Unix shell
Influenced	Python, PHP, Ruby, ECMAScript, Dao, Windows PowerShell, JavaScript
OS	Cross-platform
License	GNU General Public License, Artistic License
Website	http://www.perl.org/

- 7.2.3 Obfuscation
- 7.2.4 Poetry
- 7.2.5 CPAN Acme
- 8 Further reading
- 9 See also
- 10 References
- 11 External links

History

Larry Wall began work on Perl in 1987, while working as a programmer at Unisys,^[6] and released version 1.0 to the comp.sources.misc newsgroup on December 18, 1987.^[7] The language expanded rapidly over the next few years. Perl 2, released in 1988, featured a better regular expression engine. Perl 3, released in 1989, added support for binary data streams.

Originally the only documentation for Perl was a single (increasingly lengthy) man page. In 1991, *Programming perl* (known to many Perl programmers as the "Camel Book") was published and became the *de facto* reference for the language. At the same time, the Perl version number was bumped to 4—not to mark a major change in the language but to identify the version that was documented by the book.

Perl 4 went through a series of maintenance releases, culminating in Perl 4.036 in 1993. At that point, Wall abandoned Perl 4 to begin work on Perl 5.

Initial design of Perl 5 continued into 1994. The *perl5-porters* mailing list was established in May 1994 to coordinate work on porting Perl 5 to different platforms. It remains the primary forum for development, maintenance, and porting of Perl 5.^[8]

Perl 5 was released on October 17, 1994. It was a nearly complete rewrite of the interpreter, and it added many new features to the language, including objects, references, lexical (my) variables, and modules. Importantly, modules provided a mechanism for extending the language without modifying the interpreter. This allowed the core interpreter to stabilize, even as it enabled ordinary Perl programmers to add new language features.

As of 2009, Perl 5 is still being actively maintained. Important features and some essential new language constructs—including Unicode support, threads, improved support for object oriented programming, and many other enhancements—have been added along the way.

On December 18, 2007, the 20th anniversary of Perl 1.0, Perl 5.10.0 was released. Perl 5.10.0 includes notable new features, which bring it closer to Perl 6. Some of these new features are a new switch statement (called "given"/"when"), regular expressions updates, and the so-called smart match operator, "`~~`".^[9]

One of the most important events in Perl 5 history took place outside of the language proper and was a consequence of its module support. On October 26, 1995, the Comprehensive Perl Archive Network (CPAN) was established as a repository for Perl modules and Perl itself. At the time of writing, it carries more than 15,000 modules by more than 7,000 authors. CPAN is widely regarded as one of the greatest strengths of Perl in practice.

Name

Perl was originally named "Pearl," after the Parable of the Pearl from the Gospel of Matthew. Larry Wall wanted to give the language a short name with positive connotations; he claims that he considered (and rejected) every three- and four-letter word in the dictionary. He also considered naming it after his wife Gloria. Wall discovered the existing PEARL programming language before Perl's official release and changed the spelling of the name.

When referring to the language, the name is normally capitalized (*Perl*). When referring to the interpreter program itself, the name is often uncapitalized (*perl*) because Unix-like file systems are case-sensitive. Before the release of the first edition of *Programming Perl*, it was common to refer to the language as *perl*; Randal L. Schwartz, however, capitalised the language's name in the book to make it stand out better when typeset. This case distinction was subsequently documented as canonical.^[10]

There is contention about the all-caps spelling "PERL," which the documentation declares incorrect^[10] and which some core community members even consider a sign of outsiders.^[11] Although the name is occasionally taken as an acronym for *Practical Extraction and Report Language* (which appears at the top of the documentation^[12]), this expansion actually came after the name; several others have been suggested as equally canonical, including Wall's own humorous *Pathologically Eclectic Rubbish Lister*.^[13] Indeed, Wall claims that the name was intended to inspire many different expansions.^[14]

The camel symbol

Programming Perl, published by O'Reilly Media, features a picture of a camel on the cover and is commonly referred to as *The Camel Book*.^[6] This image of a camel has become a general symbol of Perl.

It is also a hacker emblem, appearing on some T-shirts and other clothing items.

O'Reilly owns the image as a trademark but claims to use their legal rights only to protect the "*integrity and impact of that symbol*".^[15] O'Reilly allows non-commercial use of the symbol and provides *Programming Republic of Perl* logos and *Powered by Perl* buttons.^[16] However, the Camel has never been meant to be an *official* Perl symbol, and if one is to be considered instead, it's *an onion*.^[17]

Overview

Perl is a general-purpose programming language originally developed for text manipulation and now used for a wide range of tasks including system administration, web development, network programming, and GUI development.

The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal).^[18] Its major features include support for multiple programming paradigms (procedural, object-oriented, and functional styles), reference counting memory management (without a cycle-detecting garbage collector), built-in support for text processing, and a large collection of third-party modules.

According to Larry Wall, Perl has two slogans. The first is "There's more than one way to do it," commonly known as TMTOWTDI. The second slogan is "Easy things should be easy and hard things should be possible."

Features

The overall structure of Perl derives broadly from C. Perl is procedural in nature, with variables, expressions, assignment statements, brace-delimited code blocks, control structures, and subroutines.

Perl also takes features from shell programming. All variables are marked with leading sigils, which unambiguously identify the data type (for example, scalar, array, hash) of the variable in context. Importantly, sigils allow variables to be interpolated directly into strings. Perl has many built-in functions that provide tools often used in shell programming (although many of these tools are implemented by programs external to the shell) such as sorting, and calling on system facilities.

Perl takes lists from Lisp, associative arrays (hashes) from AWK, and regular expressions from sed. These simplify and facilitate many parsing, text-handling, and data-management tasks.

In Perl 5, features were added that support complex data structures, first-class functions (that is, closures as values), and an object-oriented programming model. These include references, packages, class-based method dispatch, and lexically scoped variables, along with compiler directives (for example, the `strict` pragma). A major additional feature introduced with Perl 5 was the ability to package code as reusable modules. Larry Wall later stated that "The whole intent of Perl 5's module system was to encourage the growth of Perl culture rather than the Perl core."^[19]

All versions of Perl do automatic data typing and memory management. The interpreter knows the type and storage requirements of every data object in the program; it allocates and frees storage for them as necessary using reference counting (so it cannot de-allocate circular data structures without manual intervention). Legal type conversions—for example, conversions from number to string—are done automatically at run time; illegal type conversions are fatal errors.

Design

The design of Perl can be understood as a response to three broad trends in the computer industry: falling hardware costs, rising labor costs, and improvements in compiler technology. Many earlier computer languages, such as Fortran and C, were designed to make efficient use of expensive computer hardware. In contrast, Perl is designed to make efficient use of expensive computer programmers.

Perl has many features that ease the programmer's task at the expense of greater CPU and memory requirements. These include automatic memory management; dynamic typing; strings, lists, and hashes; regular expressions; introspection; and an `eval()` function.

Wall was trained as a linguist, and the design of Perl is very much informed by linguistic principles. Examples include Huffman coding (common constructions should be short), good end-weighting (the important information should come first), and a large collection of language primitives. Perl favors language constructs that are concise and natural for humans to read and write, even where they complicate the Perl interpreter.

Perl syntax reflects the idea that "things that are different should look different." For example, scalars, arrays, and hashes have different leading sigils. Array indices and hash keys use different kinds of braces. Strings and regular expressions have different standard delimiters. This approach can be contrasted with languages such as Lisp, where the same S-expression construct and basic syntax are used for many different purposes.

Perl does not enforce any particular programming paradigm (procedural, object-oriented, functional, and others) or even require the programmer to choose among them.

There is a broad practical bent to both the Perl language and the community and culture that surround it. The preface to *Programming Perl* begins, "Perl is a language for getting your job done." One consequence of this is that Perl is not a tidy language. It includes many features, tolerates exceptions to its rules, and employs heuristics to resolve syntactical ambiguities. Because of the forgiving nature of the compiler, bugs can sometimes be hard to find. Discussing the variant behaviour of built-in functions in list and scalar contexts, the `perlfunc(1)` manual page says, "In general, they do what you want, unless you want consistency."

In addition to Larry Wall's two slogans mentioned above, Perl has several mottos that convey aspects of its design and use, including "*Perl: the Swiss Army Chainsaw of Programming Languages*" and "*No unnecessary limits*". Perl has also been called "*The Duct Tape of the Internet*".^[20]

No written specification or standard for the Perl language exists, and there are no plans to create one for the current version of Perl. There has been only one implementation of the interpreter, and the language has evolved along with it. That interpreter, together with its functional tests, stands as a *de facto* specification of the language.

Applications

Perl has many and varied applications, compounded by the availability of many standard and third-party modules.

Perl has been used since the early days of the Web to write CGI scripts. It is known as one of "the three Ps" (along with Python and PHP), the most popular dynamic languages for writing Web applications. It is also an integral component of the popular LAMP solution stack for web development. Large projects written in Perl include Slash, Bugzilla, RT, TWiki, and Movable Type. Many high-traffic websites use Perl extensively. Examples include bbc.co.uk, Amazon.com, [LiveJournal](http://LiveJournal.com), [Ticketmaster](http://Ticketmaster.com), [Slashdot](http://Slashdot.org), [Craigslist](http://Craigslist.org), and [IMDb](http://IMDb.com).^[21]

Perl is often used as a glue language, tying together systems and interfaces that were not specifically designed to interoperate, and for "data munging", that is, converting or processing large amounts of data for tasks such as creating reports. In fact, these strengths are intimately linked. The combination makes Perl a popular all-purpose language for system administrators, particularly because short programs can be entered and run on a single command line.

With a degree of care, Perl code can be made portable across Windows and Unix. Portable Perl code is often used by suppliers of software (both COTS and bespoke) to simplify packaging and maintenance of software build and deployment scripts.

Graphical user interfaces (GUIs) may be developed using Perl. In particular, Perl/Tk is commonly used to

enable user interaction with Perl scripts. Such interaction may be synchronous or asynchronous using callbacks to update the GUI. For more information about the technologies involved, see Tk, Tcl, and WxPerl.

Perl is also widely used in finance and bioinformatics, where it is valued for rapid application development and deployment and for its capability to handle large data sets.

Implementation

Perl is implemented as a core interpreter, written in C, together with a large collection of modules, written in Perl and C. The source distribution is, as of 2005, 12 MB when packaged in a tar file and compressed. The interpreter is 150,000 lines of C code and compiles to a 1 MB executable on typical machine architectures. Alternatively, the interpreter can be compiled to a link library and embedded in other programs. There are nearly 500 modules in the distribution, comprising 200,000 lines of Perl and an additional 350,000 lines of C code. (Much of the C code in the modules consists of character-encoding tables.)

The interpreter has an object-oriented architecture. All of the elements of the Perl language—scalars, arrays, hashes, coderefs, file handles—are represented in the interpreter by C structs. Operations on these structs are defined by a large collection of macros, typedefs, and functions; these constitute the Perl C API. The Perl API can be bewildering to the uninitiated, but its entry points follow a consistent naming scheme, which provides guidance to those who use it.

The life of a Perl interpreter divides broadly into a compile phase and a run phase.^[22] In Perl, the **phases** are the major stages in the interpreter's life cycle. Each interpreter goes through each phase only once, and the phases follow in a fixed sequence.

Most of what happens in Perl's compile phase is compilation, and most of what happens in Perl's run phase is execution, but there are significant exceptions. Perl makes important use of its capability to execute Perl code during the compile phase. Perl will also delay compilation into the run phase. The terms that indicate the kind of processing that is actually occurring at any moment are **compile time** and **run time**. Perl is in compile time at most points during the compile phase, but compile time may also be entered during the run phase. The compile time for code in a string argument passed to the `eval` built-in occurs during the run phase. Perl is often in run time during the compile phase and spends most of the run phase in run time. Code in `BEGIN` blocks executes at run time but in the compile phase.

At compile time, the interpreter parses Perl code into a syntax tree. At run time, it executes the program by walking the tree. Text is parsed only once, and the syntax tree is subject to optimization before it is executed, so that execution is relatively efficient. Compile-time optimizations on the syntax tree include constant folding and context propagation, but peephole optimization is also performed.

Perl has a Turing-complete grammar because parsing can be affected by run-time code executed during the compile phase.^[23] Therefore, Perl cannot be parsed by a straight Lex/Yacc lexer/parser combination. Instead, the interpreter implements its own lexer, which coordinates with a modified GNU bison parser to resolve ambiguities in the language.

It is often said that "Only perl can parse Perl," meaning that only the Perl interpreter (*perl*) can parse the Perl language (*Perl*), but even this is not, in general, true. Because the Perl interpreter can simulate a Turing machine during its compile phase, it would need to decide the Halting Problem in order to complete parsing in every case. It's a long-standing result that the Halting Problem is undecidable, and therefore not even Perl can always parse Perl. Perl makes the unusual choice of giving the user access to its full programming power in its own compile phase. The cost in terms of theoretical purity is high, but practical inconvenience seems to be rare.

Other programs that undertake to parse Perl, such as source-code analyzers and auto-indenters, have to contend not only with ambiguous syntactic constructs but also with the undecidability of Perl parsing in the general case. Adam Kennedy's PPI project focused on parsing Perl code as a document (retaining its integrity as a document), instead of parsing Perl as executable code (which not even Perl itself can always do). It was Kennedy who first conjectured that, "parsing Perl suffers from the 'Halting Problem'".^[24] and this was later proved.^[25]

Perl is distributed with some 120,000 functional tests. These run as part of the normal build process and extensively exercise the interpreter and its core modules. Perl developers rely on the functional tests to ensure that changes to the interpreter do not introduce bugs; conversely, Perl users who see that the interpreter passes its functional tests on their system can have a high degree of confidence that it is working properly.

Maintenance of the Perl interpreter has become increasingly difficult over the years. The code base has been in continuous development since 1994. The code has been optimized for performance at the expense of simplicity, clarity, and strong internal interfaces. New features have been added, yet virtually complete backward compatibility with earlier versions is maintained. The size and complexity of the interpreter is a barrier to developers who wish to work on it.

Availability

Perl is free software and is licensed under both the Artistic License and the GNU General Public License. Distributions are available for most operating systems. It is particularly prevalent on Unix and Unix-like systems, but it has been ported to most modern (and many obsolete) platforms. With only six reported exceptions, Perl can be compiled from source code on all Unix-like, POSIX-compliant, or otherwise-Unix-compatible platforms.^[26] However, this is rarely necessary, because Perl is included in the default installation of many popular operating systems.

Because of unusual changes required for the Mac OS Classic environment, a special port called MacPerl was shipped independently.^[27]

The CPAN carries a complete list of supported platforms with links to the distributions available on each.^[28]

Windows

Users of Microsoft Windows typically install one of the native binary distributions of Perl for Win32^[29], most commonly ActivePerl. Compiling Perl from source code under Windows is possible, but most installations lack the requisite C compiler and build tools. This also makes it difficult to install modules from the CPAN, particularly those that are partially written in C.

Users of the ActivePerl binary distribution are, therefore, dependent on the repackaged modules provided in ActiveState's module repository, which are precompiled and can be installed with PPM. Limited resources to maintain this repository have been cause for various long-standing problems.^{[30][31]}

To address this and other problems of Perl on the Windows platform, win32.perl.org (<http://win32.perl.org/>) was launched by Adam Kennedy on behalf of The Perl Foundation in June 2006. This is a community website for "all things Windows and Perl." A major aim of this project is to provide production-quality alternative Perl distributions that include an embedded C compiler and build tools, so as to enable Windows users to install modules directly from the CPAN. The production distribution in the family is known as Strawberry Perl (<http://strawberryperl.com/>) , with research and experimental work done in a related Vanilla Perl (<http://vanillaperl.com/>) distribution.

Another popular way of running Perl under Windows is provided by the Cygwin emulation layer. Cygwin provides a Unix-like environment on Windows, and both perl and cpan are conveniently available as standard pre-compiled packages in the Cygwin setup program. Because Cygwin also includes the gcc, compiling Perl from source is also possible.

Language structure

In Perl, the minimal Hello world program may be written as follows:

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

This prints the string *Hello, world!* and a newline, symbolically expressed by an `n` character whose interpretation is altered by the preceding escape character (a backslash).

The canonical form of the program is slightly more verbose:

```
#!/usr/bin/perl
print "Hello, world!\n";
```

The hash mark character introduces a comment in Perl, which runs up to the end of the line of code and is ignored by the compiler. The comment used here is of a special kind: it's called the shebang line. This tells

Unix-like operating systems where to find the Perl interpreter, making it possible to invoke the program without explicitly mentioning `perl`. (Note that, on Microsoft Windows systems, Perl programs are typically invoked by associating the `.pl` extension with the Perl interpreter. In order to deal with such circumstances, `perl` detects the shebang line and parses it for switches^[32]; therefore, it is not strictly true that the shebang line is ignored by the compiler.)

The second line in the canonical form includes a semicolon, which is used to separate statements in Perl. With only a single statement in a block or file, a separator is unnecessary, so it can be omitted from the minimal form of the program—or more generally from the final statement in any block or file. The canonical form includes it because it is common to terminate every statement even when it is unnecessary to do so, as this makes editing easier: code can be added to, or moved away from, the end of a block or file without having to adjust semicolons.

Version 5.10 of Perl introduces a `say` function that implicitly appends a newline character to its output, making the minimal "Hello world" program even shorter:

```
say 'Hello, world!'
```

Data types

Perl has a number of fundamental data types. The most commonly used and discussed are scalars, arrays, hashes, filehandles, and subroutines:

- A scalar is a single value; it may be a number, a string, or a reference.
- An array is an ordered collection of scalars.
- A hash, or associative array, is a map from strings to scalars; the strings are called *keys*, and the scalars are called *values*.
- A file handle is a map to a file, device, or pipe that is open for reading, writing, or both.
- A subroutine is a piece of code that may be passed arguments, be executed, and return data

Most variables are marked by a leading sigil, which identifies the data type being accessed (not the type of the variable itself), except filehandles, which don't have a sigil. The same name may be used for variables of different data types, without conflict.

```
$foo # a scalar
@foo # an array
%foo # a hash
$FOO # a file handle
&foo # a subroutine (but the & is optional)
```

File handles and constants need not be uppercase, but it is a common convention because there is no sigil to denote them. Both are global in scope, but file handles are interchangeable with references to file handles, which can be stored in scalars, which in turn permit lexical scoping. Doing so is encouraged in Damian Conway's *Perl Best Practices*. As a convenience, the `open` function in Perl 5.6 and newer will autovivify undefined scalars to file handle references.

Numbers are written in the bare form; strings are enclosed by quotes of various kinds.

```

$name = "joe";
$color = 'red';

$number1 = 42;
$number2 = '42';

# This evaluates to true
if ($number1 == $number2) { print "Numbers and strings of numbers are the same!"; }

$answer = "The answer is $number1"; # Variable interpolation: The answer is 42
$price = 'This device costs $42'; # No interpolation in single quotes

$album = "It's David Bowie's \"Heroes\""; # literal quotes inside a string;
$album = 'It\'s David Bowie\'s "Heroes"'; # same as above with single quotes;
$album = q(It's David Bowie's "Heroes"); # the quote-like operators q() and qq() allow
# almost any delimiter instead of quotes, to
# avoid excessive backslashing

$multilined_string =<<EOF;
This is my multilined string
note that I am terminating it with the "EOF" word.
EOF

```

Perl will convert strings into numbers and vice versa depending on the context in which they are used. In the following example, the strings `$n` and `$m` are treated as numbers when they are the arguments to the addition operator. This code prints the number '5', discarding non-numeric information for the operation, although the variable values remain the same. (The string concatenation operator is the period, not the `+` symbol.)

```

$n = '3 apples';
$m = '2 oranges';
print $n + $m;

```

Perl also has a boolean context that it uses in evaluating conditional statements. The following values all evaluate as false in Perl:

```

>false = 0; # the number zero
>false = 0.0; # the number zero as a float
>false = 0b0; # the number zero in binary
>false = 0x0; # the number zero in hexadecimal
>false = '0'; # the string zero
>false = ""; # the empty string
>false = undef; # the return value from undef

```

All other values are evaluated to true. This includes the odd self-describing literal string of "0 but true," which in fact is 0 as a number, but true when used as a boolean. (Any non-numeric string would also have this property, but this particular string is ignored by Perl with respect to numeric warnings.) A less-explicit but more conceptually portable version of this string is '0E0' or '0e0', which does not rely on characters being evaluated as 0, because '0E0' is literally "zero times ten to the zeroth power."

Evaluated boolean expressions also return scalar values. Although the documentation does not promise which *particular* true or false is returned (and thus cannot be relied on), many boolean operators return 1 for true and the empty-string for false (which evaluates to zero in a numeric context). The *defined()* function tells if the variable has any value set. In the above examples, *defined(\$false)* is true for every value except *undef*.

If a specifically 1 or 0 result (as in C) is needed, an explicit conversion is thought by some authors to be required:

```

my $real_result = $boolean_result ? 1 : 0;

```

However, if it's known that the value is either 1 or *undef*, an implicit conversion can be used instead:

```

my $real_result = $boolean_result + 0;

```

A list is written by listing its elements, separated by commas, and enclosed by parentheses where required by operator precedence.

```

@scores = (32, 45, 16, 5);

```


It can be written many other ways as well, some straightforward and some less so:

```
# An explicit and straightforward way
@scores = ('32', '45', '16', '5');

# Equivalent to the above, but the qw() quote-like operator saves typing of
# quotes and commas and reduces visual clutter; almost any delimiter can be
# used instead of parentheses
@scores = qw(32 45 16 5);

# The split function returns a list of strings, which are extracted
# from the expression using a regex template.
# This may be useful for reading from a file of comma-separated values (CSV)
@scores = split /,/, '32,45,16,5';

# It's also possible to use a postfix for operator and aliasing of
# the $_ magic variable to the next value of the list during each
# iteration; this is pointless here, but similar idioms are widely used
# in some circumstances.
push @scores, $_ foreach 32, 45, 16, 5;
```

A hash may be initialized from a list of key/value pairs:

```
%favorite = (
    joe => 'red',
    sam => 'blue'
);
```

The => operator is equivalent to a comma, except that it assumes quotes around the preceding token if it is a bare identifier: (joe => 'red') is the same as ('joe' => 'red'). It can therefore be used to elide quote marks, improving readability.

Individual elements of a list are accessed by providing a numerical index, in square brackets. Individual values in a hash are accessed by providing the corresponding key, in curly braces. The \$ sigil identifies the accessed element as a scalar.

```
$scores[2] # an element of @scores
$favorite{joe} # a value in %favorite
```

Thus, a hash can also be specified by setting its keys individually:

```
$favorite{joe} = 'red';
$favorite{sam} = 'blue';
```

Multiple elements may be accessed by using the @ sigil instead (identifying the result as a list).

```
@scores[2, 3, 1] # three elements of @scores
@favorite{'joe', 'sam'} # two values in %favorite
@favorite{qw(joe sam)} # same as above
```

The number of elements in an array can be obtained by evaluating the array in scalar context or with the help of the \$# sigil. The latter gives the index of the last element in the array, not the number of elements. *Note: the syntax highlighting in Wikipedia's software mistakenly considers some of the following code to be part of the comments.*

```
$count = @friends; # Assigning to a scalar forces scalar context

$#friends; # The index of the last element in @friends
$#friends+1; # Usually the number of elements in @friends is one more
# than $#friends because the first element is at index 0,
# not 1, unless the programmer reset this to a different
# value, which most Perl manuals discourage.
```

There are a few functions that operate on entire hashes.

```

@names = keys %addressbook;
@addresses = values %addressbook;

# Every call to each returns the next key/value pair.
# All values will be eventually returned, but their order
# cannot be predicted.
while (($name, $address) = each %addressbook) {
    print "$name lives at $address\n";
}

# Similar to the above, but sorted alphabetically
foreach my $next_name (sort keys %addressbook) {
    print "$next_name lives at $addressbook{$next_name}\n";
}

```

Control structures

Perl has several kinds of control structures.

It has block-oriented control structures, similar to those in the C, Javascript, and Java programming languages. Conditions are surrounded by parentheses, and controlled blocks are surrounded by braces:

```

label while ( cond ) { ... }
label while ( cond ) { ... } continue { ... }
label for ( init-expr ; cond-expr ; incr-expr ) { ... }
label foreach var ( list ) { ... }
label foreach var ( list ) { ... } continue { ... }
if ( cond ) { ... }
if ( cond ) { ... } else { ... }
if ( cond ) { ... } elsif ( cond ) { ... } else { ... }

```

Where only a single statement is being controlled, statement modifiers provide a more-concise syntax:

```

statement if cond ;
statement unless cond ;
statement while cond ;
statement until cond ;
statement foreach list ;

```

Short-circuit logical operators are commonly used to affect control flow at the expression level:

```

expr and expr
expr && expr
expr or expr
expr || expr

```

(The "and" and "or" operators are similar to && and || but have lower precedence, which makes it easier to use them to control entire statements.)

The flow control keywords `next` (corresponding to C's `continue`), `last` (corresponding to C's `break`), `return`, and `redo` are expressions, so they can be used with short-circuit operators.

Perl also has two implicit looping constructs, each of which has two forms:

```

$results = grep { ... } list
$results = grep expr, list
$results = map { ... } list
$results = map expr, list

```

`grep` returns all elements of *list* for which the controlled block or expression evaluates to true. `map` evaluates the controlled block or expression for each element of *list* and returns a list of the resulting values. These constructs enable a simple functional programming style.

Up until the 5.10.0 release, there was no switch statement in Perl 5. From 5.10.0 onward, a multi-way branch statement called `given/when` is available, which takes the following form:

```

given ( expr ) { when ( cond ) { ... } default { ... } }

```

Syntactically, this structure behaves similarly to switch statements found in other languages, but with a few important differences. The largest is that unlike switch/case structures, given/when statements break execution after the first successful branch, rather than waiting for explicitly defined break commands. Conversely, explicit continues are instead necessary to emulate switch behavior.

For those not using the 5.10.0 release, the Perl documentation describes a half-dozen ways to achieve the same effect by using other control structures. There is also a Switch (<http://search.cpan.org/perldoc/Switch>) module, which provides functionality modeled on the forthcoming Perl 6 re-design. It is implemented using a source filter, so its use is unofficially discouraged.^[33]

Perl includes a `goto label` statement, but it is rarely used. Situations where a `goto` is called for in other languages don't occur as often in Perl because of its breadth of flow control options.

There is also a `goto &sub` statement that performs a tail call. It terminates the current subroutine and immediately calls the specified *sub*. This is used in situations where a caller can perform more-efficient stack management than Perl itself (typically because no change to the current stack is required), and in deep recursion, tail calling can have substantial positive impact on performance because it avoids the overhead of scope/stack management on return.

Subroutines

Subroutines are defined with the `sub` keyword and are invoked simply by naming them. If the subroutine in question has not yet been declared, invocation requires either parentheses after the function name or an ampersand (`&`) before it. But using `&` without parentheses will also implicitly pass the arguments of the current subroutine to the one called, and using `&` with parentheses will bypass prototypes.

```
# Calling a subroutine
# Parentheses are required here if the subroutine is defined later in the code
foo();
&foo; # (this also works, but has other consequences regarding arguments passed to the subroutine)

# Defining a subroutine
sub foo { ... }

foo; # Here parentheses are not required
```

A list of arguments may be provided after the subroutine name. Arguments may be scalars, lists, or hashes.

```
foo $x, @y, %z;
```

The parameters to a subroutine do not need to be declared as to either number or type; in fact, they may vary from call to call. Any validation of parameters must be performed explicitly inside the subroutine.

Arrays are expanded to their elements; hashes are expanded to a list of key/value pairs; and the whole lot is passed into the subroutine as one flat list of scalars.

Whatever arguments are passed are available to the subroutine in the special array `@_`. The elements of `@_` are aliased to the actual arguments; changing an element of `@_` changes the corresponding argument.

Elements of `@_` may be accessed by subscripting it in the usual way.

```
$_[0], $_[1]
```

However, the resulting code can be difficult to read, and the parameters have pass-by-reference semantics, which may be undesirable.

One common idiom is to assign `@_` to a list of named variables.

```
my ($x, $y, $z) = @_;
```

This provides mnemonic parameter names and implements pass-by-value semantics. The `my` keyword indicates that the following variables are lexically scoped to the containing block.

Another idiom is to shift parameters off of `@_`. This is especially common when the subroutine takes only one argument or for handling the `$self` argument in object-oriented modules.

```
my $x = shift;
```

Subroutines may assign `@_` to a hash to simulate named arguments; this is recommended in *Perl Best Practices* for subroutines that are likely to ever have more than three parameters.^[34]

```
sub function1 {
    my %args = @_;
    print "'x' argument was '$args{x}'\n";
}
function1( x => 23 );
```

Subroutines may return values.

```
return 42, $x, @y, %z;
```

If the subroutine does not exit via a `return` statement, then it returns the last expression evaluated within the subroutine body. Arrays and hashes in the return value are expanded to lists of scalars, just as they are for arguments.

The returned expression is evaluated in the calling context of the subroutine; this can surprise the unwary.

```
sub list { (4, 5, 6) }
sub array { @x = (4, 5, 6); @x }

$x = list; # returns 6 - last element of list
$x = array; # returns 3 - number of elements in list
@x = list; # returns (4, 5, 6)
@x = array; # returns (4, 5, 6)
```

A subroutine can discover its calling context with the `wantarray` function.

```
sub either {
    return wantarray ? (1, 2) : 'Oranges';
}

$x = either; # returns "Oranges"
@x = either; # returns (1, 2)
```

Regular expressions

The Perl language includes a specialized syntax for writing regular expressions (RE, or regexes), and the interpreter contains an engine for matching strings to regular expressions. The regular-expression engine uses a backtracking algorithm, extending its capabilities from simple pattern matching to string capture and substitution. The regular-expression engine is derived from `regex` written by Henry Spencer.

The Perl regular-expression syntax was originally taken from Unix Version 8 regular expressions. However, it diverged before the first release of Perl and has since grown to include far more features. Many other languages and applications are now adopting Perl compatible regular expressions over POSIX regular expressions, such as PHP, Ruby, Java, Microsoft's .NET Framework^[35], and the Apache HTTP server.

Regular-expression syntax is extremely compact, owing to history. The first regular-expression dialects were only slightly more expressive than globs, and the syntax was designed so that an expression would resemble the text that it matches. This meant using no more than a single punctuation character or a pair of delimiting characters to express the few supported assertions. Over time, the expressiveness of regular expressions grew tremendously, but the syntax design was never revised and continues to rely on punctuation. As a result, regular expressions can be cryptic and extremely dense.

Uses

The `m//` (match) operator introduces a regular-expression match. (If it is delimited by slashes, as in all of the

examples here, then the leading `m` may be omitted for brevity. If the `m` is present, as in all of the following examples, other delimiters can be used in place of slashes.) In the simplest case, an expression such as

```
$x =~ /abc/;
```

evaluates to true if and only if the string `$x` matches the regular expression `abc`.

The `s///` (substitute) operator, on the other hand, specifies a search-and-replace operation:

```
$x =~ s/abc/aBc/; # upcase the b
```

Another use of regular expressions is to specify delimiters for the `split` function:

```
@words = split /,/, $line;
```

The `split` function creates a list of the parts of the string that are separated by matches of the regular expression. In this example, a line is divided into a list of its comma-separated parts, and this list is then assigned to the `@words` array.

Syntax

Portions of a regular expression may be enclosed in parentheses; corresponding portions of a matching string are *captured*. Captured strings are assigned to the sequential built-in variables `$1`, `$2`, `$3`, ..., and a list of captured strings is returned as the value of the match.

```
$x =~ /a(.)c/; # capture the character between 'a' and 'c'
```

Perl regular expressions can take *modifiers*. These are single-letter suffixes that modify the meaning of the expression:

```
$x =~ /abc/i; # case-insensitive pattern match  
$x =~ s/abc/aBc/g; # global search and replace
```

Because the compact syntax of regular expressions can make them dense and cryptic, the `/x` modifier was added in Perl to help programmers write more-legible regular expressions. It allows programmers to place whitespace and comments *inside* regular expressions:

```
$x =~ /a # match 'a'  
    . # followed by any character  
    c # then followed by the 'c' character  
/x;
```

Database interfaces

Perl is widely favored for database applications. Its text-handling facilities are useful for generating SQL queries; arrays, hashes, and automatic memory management make it easy to collect and process the returned data.

In early versions of Perl, database interfaces were created by relinking the interpreter with a client-side database library. This was sufficiently difficult that it was done for only a few of the most-important and most widely used databases, and it restricted the resulting perl executable to using just one database interface at a time.

In Perl 5, database interfaces are implemented by Perl DBI modules. The DBI (Database Interface) module presents a single, database-independent interface to Perl applications, while the DBD (Database Driver) modules handle the details of accessing some 50 different databases; there are DBD drivers for most ANSI SQL databases.

DBI provides caching for database handles and queries, which can greatly improve performance in long-lived

execution environments such as `mod_perl`^[36], helping high-volume systems avert load spikes as in the Slashdot effect.

Comparative performance

The Computer Language Benchmarks Game^[37] compares the performance of implementations of typical programming problems in several programming languages. The submitted Perl implementations were typically toward the high end of the memory-usage spectrum and had varied speed results. Perl's performance in the benchmarks game is typical for interpreted languages and places it toward the lead in that group.

Large Perl programs start slower than similar programs in compiled languages because perl has to compile the source every time it runs. In a talk at the YAPC::Europe 2005 conference and subsequent article "A Timely Start," Jean-Louis Leroy found that his Perl programs took much longer to run than he expected because the perl interpreter spent much of the time finding modules because of his over-large include path.^[38] Unlike Java, Python, and Ruby, Perl has only experimental support for pre-compiling.^[39] Therefore Perl programs pay this overhead penalty on every execution. The run phase of typical programs is long enough that amortized startup time is not substantial, but results in benchmarks that measure very short execution times are likely to be skewed.

A number of tools have been introduced to improve this situation. The first such tool was Apache's `mod_perl`, which sought to address one of the most-common reasons that small Perl programs were invoked rapidly: CGI Web development. ActivePerl, via Microsoft ISAPI, provides similar performance improvements.

Once Perl code is compiled, there is additional overhead during the execution phase that typically isn't present for programs written in compiled languages such as C or C++. Examples of such overhead include bytecode interpretation, reference-counting memory management, and dynamic type checking.

Optimizing

Like any code, Perl programs can be tuned for performance using benchmarks and profiles after a readable and correct implementation is finished. In part because of Perl's interpreted nature, writing more-efficient Perl will not always be enough to meet one's performance goals for a program.

In such situations, the most-critical routines of a Perl program can be written in other languages such as C or Assembler, which can be connected to Perl via simple Inline modules or the more-complex-but-flexible XS mechanism.^[40] Nicholas Clark, a Perl core developer, discusses some Perl design trade-offs and some solutions in *When perl is not quite fast enough*.^[41]

In extreme cases, optimizing Perl can require intimate knowledge of the interpreter's workings rather than skill with algorithms, the Perl language, or general principles of optimization.

Future

At the 2000 Perl Conference, Jon Orwant made a case for a major new language initiative.^[42] This led to a decision to begin work on a redesign of the language, to be called Perl 6. Proposals for new language features were solicited from the Perl community at large, and more than 300 RFCs were submitted.

Larry Wall spent the next few years digesting the RFCs and synthesizing them into a coherent framework for Perl 6. He has presented his design for Perl 6 in a series of documents called "apocalypses," which are numbered to correspond to chapters in *Programming Perl* ("The Camel Book"). The current, not-yet-finalized specification of Perl 6 is encapsulated in design documents called Synopses, which are numbered to correspond to Apocalypses.

Perl 6 is not intended to be backward compatible, although there will be a compatibility mode.

Thesis work by Bradley M. Kuhn, overseen by Larry Wall, considered the possible use of the Java virtual machine as a runtime for Perl.^[43] Kuhn's thesis showed this approach to be problematic, and in 2001, it was decided that Perl 6 would run on a cross-language virtual machine called Parrot. This will mean that other languages targeting the Parrot will gain native access to CPAN, allowing some level of cross-language development.

In 2005, Audrey Tang created the pugs project, an implementation of Perl 6 in Haskell. This was, and continues to act as, a test platform for the Perl 6 language (separate from the development of the actual implementation) allowing the language designers to explore. The pugs project spawned an active Perl/Haskell cross-language community centered around the freenode `#perl6` irc channel.

A number of features in the Perl 6 language now show similarities to Haskell, and Perl 6 has been embraced by the Haskell community as a potential scripting language.

As of 2006, Perl 6, Parrot, and pugs are under active development, and a new module for Perl 5 called `v6` allows some Perl 6 code to run directly on top of Perl 5.

Development of Perl 5 is also continuing. Perl 5.10 was released in December 2007, with some new features influenced by the design of Perl 6.

The Perl community

Perl's culture and community has developed alongside the language itself. Usenet was the first public venue in which Perl was introduced, but over the course of its evolution, Perl's community was shaped by the growth of broadening Internet-based services including the introduction of the World Wide Web. The community that surrounds Perl was, in fact, the topic of Larry Wall's first "State of the Onion" talk.^[44]

State of the Onion

State of the Onion is the name for Larry Wall's yearly keynote-style summaries on the progress of Perl and its community. They are characterized by his hallmark humor, employing references to Perl's culture, the wider hacker culture, Wall's linguistic background, sometimes his family life, and occasionally even his Christian background.

Each talk is first given at various Perl conferences and is eventually also published online.

Pastimes

Perl's pastimes have become a defining element of the community. Included among them are trivial and complex uses of the language.

JAPHs

In email, Usenet, and message-board postings, "Just another Perl hacker" (JAPH) programs have become a common trend, originated by Randal L. Schwartz, one of the earliest professional Perl trainers.^[45]

In the parlance of Perl culture, Perl programmers are known as Perl hackers, and from this derives the practice of writing short programs to print out the phrase "Just another Perl hacker,". In the spirit of the original concept, these programs are moderately obfuscated and short enough to fit into the signature of an email or Usenet message. The "canonical" JAPH includes the comma at the end, although this is often omitted.

Perl golf

Perl "golf" is the pastime of reducing the number of characters (key "strokes") used in a Perl program to the bare minimum, much as how golf players seek to take as few shots as possible in a round. This use of the word "golf" originally focused on the JAPHs used in signatures in Usenet postings and elsewhere, although the same stunts had been an unnamed pastime in the language APL in previous decades. The use of Perl to write a program that performed RSA encryption prompted a widespread and practical interest in this pastime.^[46] In subsequent years, the term "code golf" has been applied to the pastime in other languages.^[47]

Obfuscation

As with C, obfuscated code competitions are a well-known pastime. The annual Obfuscated Perl contest made an arch virtue of Perl's syntactic flexibility.

Poetry

Similar to obfuscated code and golf, but with a different purpose, Perl poetry is the practice of writing poems that can actually be compiled as legal (although generally non-sensical) Perl code. This hobby is more or less unique to Perl because of the large number of regular English words that are used in the language. New poems are regularly published in the Perl Monks site's Perl Poetry section.^[48]

CPAN Acme

There are also many examples of code written purely for entertainment on the CPAN. `Lingua::Romana::Perligata`, for example, allows writing programs in Latin.^[49] Upon execution of such a program, the module translates its source code into regular Perl and runs it.

The Perl community has set aside the "Acme" namespace for modules that are fun in nature (but its scope has widened to include exploratory or experimental code or any other module that is not meant to ever be used in production). Some of the Acme modules are deliberately implemented in amusing ways. This includes

`Acme::Bleach`, one of the first modules in the `Acme::` namespace,^[50] which allows the program's source code to be "whitened" (i.e., all characters replaced with whitespace) and yet still work.

Further reading

- *Learning Perl*, Fifth Edition (the *Llama book*), ISBN 0-596-52010-6
- *Perl Cookbook*, ISBN 0-596-00313-7
- *Programming Perl* (the *Camel book*), ISBN 0-596-00027-8

See also

- Autovivification
- Comparison of programming languages
- Just another Perl hacker
- Perl Data Language
- Perl Object Environment
- PerlScript
- Plain Old Documentation



References

1. ^ What is Perl? (<http://perl.about.com/od/gettingstartedwithperl/p/whatisperl.htm>)
2. ^ Beginner's Introduction to Perl (<http://www.perl.com/pub/a/2000/10/begperl1.html>)
3. ^ Ashton, Elaine (1999). "The Timeline of Perl and its Culture (v3.0_0505)". <http://history.perl.org/PerlTimeline.html>.
4. ^ Wall, Larry, Tom Christiansen and Jon Orwant (July 2000). *Programming Perl, Third Edition*. O'Reilly. ISBN 0-596-00027-8.
5. ^ Sheppard, Doug (2000-10-16). "Beginner's Introduction to Perl". O'Reilly Media. <http://www.perl.com/pub/a/2000/10/begperl1.html>. Retrieved on 2008-07-27.
6. ^ ^a ^b "Larry Wall". http://www.perl.com/pub/a/Wall_Larry. Retrieved on 2006-08-20.
7. ^ "Perl, a "replacement" for awk and sed". http://groups.google.com/group/comp.sources.unix/browse_thread/thread/363c7a6fa4e2668b/bb3ee125385ae25f. Retrieved on 2007-12-18.
8. ^ perl5-porters archive (<http://www.nntp.perl.org/group/perl.perl5.porters/>)
9. ^ perldelta: what is new for perl 5.10.0 (<http://search.cpan.org/~rgarcia/perl-5.10.0-RC2/pod/perl5100delta.pod>)
10. ^ ^a ^b "perlfaq1: What's the difference between "perl" and "Perl"?". <http://perldoc.perl.org/perlfaq1.html#What's-the-difference-between-%22perl%22-and-%22Perl%22%3f>.
11. ^ Schwartz, Randal. "PERL as shibboleth and the Perl community". http://www.perlmonks.org/index.pl?node_id=510594. Retrieved on 2007-06-01.
12. ^ Wall, Larry. "Larry Wall". <http://www.linuxjournal.com/article/3394>. Retrieved on 2008-10-02.
13. ^ Wall, Larry. "BUGS". *perl(1) man page*. <http://perldoc.perl.org/perl.html#BUGS>. Retrieved on 2006-10-13.
14. ^ Wall, Larry. "Re^7: PERL as shibboleth and the Perl community". http://www.perlmonks.org/index.pl?node_id=511722. Retrieved on 2007-01-03.
15. ^ O'Reilly—The Perl Camel Usage and Trademark Information (<http://perl.oreilly.com/usage/>)
16. ^ Index of /images/perl (<http://www.oreillynet.com/images/perl/>)
17. ^ Perl Trademark, User Logos, Perl Marks and more (http://www.perlfoundation.org/perl_trademark)
18. ^ perlintro(1) man page
19. ^ Usenet post, May 10th 1997, with ID 199705101952.MAA00756@wall.org.

20. ^ "The Importance of Perl". O'Reilly & Associates, Inc.. April 1998. http://www.oreillynet.com/pub/a/oreilly/perl/news/importance_0498.html. "As Hassan Schroeder, Sun's first webmaster, remarked: "Perl is the duct tape of the Internet.""
21. ^ "IMDb Helpdesk: What software/hardware are you using to run the site?". http://www.imdb.com/help/search?domain=helpdesk_faq&index=1&file=techinfo. Retrieved on 2007-09-01.
22. ^ A description of the Perl 5 interpreter can be found in *Programming Perl*, 3rd Ed., chapter 18 (<http://www.oreilly.com/catalog/ppperl3/chapter/ch18.html>) . See particularly page 467, which carefully distinguishes run phase and compile phase from run time and compile time. Perl "time" and "phase" are often confused.
23. ^ Schwartz, Randal. "On Parsing Perl". http://www.perlmonks.org/index.pl?node_id=44722. Retrieved on 2007-01-03.
24. ^ The quote is from Kennedy, Adam (2006). "PPI—Parse, Analyze and Manipulate Perl (without perl)". CPAN. <http://search.cpan.org/~adamk/PPI-1.201/lib/PPI.pm>.
25. ^ "Rice's Theorem". *The Perl Review* **4** (3): 23-29. Summer 2008. and "Perl is Undecidable". *The Perl Review* **5** (0): 7-11. Fall 2008., which is available online at Kegler, Jeffrey. "Perl and Undecidability". <http://www.jeffreykegler.com/Home/perl-and-undecidability>.
26. ^ Hietaniemi, Jarkko (1998). "Perl Ports (Binary Distributions)". CPAN.org. <http://www.cpan.org/ports/>.
27. ^ "The MacPerl Pages". Prime Time Freeware. 1997. <http://www.macperl.com/>.
28. ^ CPAN/ports (<http://www.cpan.org/ports/>)
29. ^ "Win32 Distributions". Win32 Perl Wiki. http://win32.perl.org/wiki/index.php?title=Win32_Distributions#Perl_Distributions
30. ^ Golden, David (2006). "Activestate and Scalar-List-Utills". <http://www.mail-archive.com/perl-qa@perl.org/msg05407.html>.
31. ^ Kennedy, Adam (2007). "ActivePerl PPM repository design flaw goes critical". <http://use.perl.org/~Alias/journal/35219>.
32. ^ "perlrun manpage". <http://perldoc.perl.org/perlrun.html#DESCRIPTION>.
33. ^ using switch (http://www.perlmonks.org/?node_id=496084)
34. ^ Damian Conway, *Perl Best Practices* (<http://www.oreilly.com/catalog/perlbp/chapter/ch09.pdf>) , p.182
35. ^ Microsoft Corp., ".NET Framework Regular Expressions", *.NET Framework Developer's Guide*, [1] ([http://msdn2.microsoft.com/en-us/library/hs600312\(VS.71\).aspx](http://msdn2.microsoft.com/en-us/library/hs600312(VS.71).aspx))
36. ^ Bekman, Stas. "Efficient Work with Databases under mod_perl". http://perl.apache.org/docs/1.0/guide/performance.html#Efficient_Work_with_Databases_u Retrieved on 2007-09-01.
37. ^ The Computer Language Benchmarks Game (<http://shootout.alioth.debian.org/>)
38. ^ Leroy, Jean-Louis (2005-12-01). "A Timely Start". Perl.com. http://www.perl.com/pub/a/2005/12/21/a_timely_start.html.
39. ^ Beattie, Malcolm and Enache Adrian (2003). "B::Bytecode Perl compiler's bytecode backend". search.cpan.org. http://search.cpan.org/~nwclark/perl-5.8.8/ext/B/B/Bytecode.pm#KNOWN_BUGS.
40. ^ <http://search.cpan.org/perldoc/Inline/>
41. ^ When perl is not quite fast enough (http://www.ccl4.org/~nick/P/Fast_Enough/)
42. ^ Transcription of Larry's talk (<http://www.nntp.perl.org/group/perl.perl6.meta/424>) . Retrieved on 2006 September 28.
43. ^ Kuhn, Bradley (January 2001). "*Considerations on Porting Perl to the Java Virtual Machine*" (<http://www.ebb.org/bkuhn/writings/technical/thesis/>) ". University of Cincinnati. Retrieved on 2008-06-28.
44. ^ Wall, Larry (1997-08-20). "Perl Culture (AKA the first State of the Onion)". <http://www.wall.org/~larry/keynote/keynote.html>.
45. ^ Randal L. Schwartz (1999-05-02). "*Who is Just another Perl hacker?*" (news:m1hfpvh2jq.fsf@halfdome.holdit.com) ". comp.lang.perl.misc (news:comp.lang.perl.misc) . (Web link) (<http://groups.google.com/groups?selm=m1hfpvh2jq.fsf@halfdome.holdit.com>) . Retrieved on 2007-11-12.
46. ^ The quest for the most diminutive munitions program (<http://www.cypherspace.org/adam/rsa/story.html>)
47. ^ "Code Golf: What is Code Golf?". 29degrees. 2007. <http://codegolf.com/>.
48. ^ Perl Poetry section (http://www.perlmonks.org/?node_id=1590) on Perl Monks
49. ^ Conway, Damian. "Lingua::Romana::Perligata -- Perl for the XXI-imum Century". <http://www.csse.monash.edu.au/~damian/papers/HTML/Perligata.html>.
50. ^ Brocard, Leon (2001-05-23). "use Perl; Journal of acme". <http://use.perl.org/~acme/journal/200>.

External links

- Perl.org (<http://www.perl.org/>) —Official Perl website
- Perl documentation (<http://perldoc.perl.org/>)
- The Perl Foundation (<http://www.perlfoundation.org/>)
- Official Perl 5 Wiki (<http://www.perlfoundation.org/perl5/>)
- Useful Perl tutorials for beginner (<http://www.perltutorial.org/>)
- Perl (<http://www.dmoz.org/Computers/Programming/Languages/Perl/>) at the Open Directory Project

Retrieved from "http://en.wikipedia.org/wiki/Perl"

Categories: Perl | Curly bracket programming languages | Dynamic programming languages |

Dynamically-typed programming languages | Free compilers and interpreters | Procedural programming languages | Scripting languages | Text-oriented programming languages | Unix software | Cross-platform software

Hidden categories: Articles containing potentially dated statements from 2009 | All articles containing potentially dated statements | Articles containing potentially dated statements from 2005 | Wikipedia articles needing style editing from December 2008 | All articles needing style editing | All articles with unsourced statements | Articles with unsourced statements since June 2007 | Articles with unsourced statements since July 2007 | Articles with unsourced statements since October 2007 | Articles containing potentially dated statements from 2006

- All text is available under the terms of the GNU Free Documentation License. (See **Copyrights** for details.)

Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a U.S. registered 501(c)(3) tax-deductible nonprofit charity.