

## NAME

perldata – Perl data types

## DESCRIPTION

**Variable names**

Perl has three data structures: scalars, arrays of scalars, and associative arrays of scalars, known as “hashes”. Normal arrays are indexed by number, starting with 0. (Negative subscripts count from the end.) Hash arrays are indexed by string.

Values are usually referred to by name (or through a named reference). The first character of the name tells you to what sort of data structure it refers. The rest of the name tells you the particular value to which it refers. Most often, it consists of a single *identifier*, that is, a string beginning with a letter or underscore, and containing letters, underscores, and digits. In some cases, it may be a chain of identifiers, separated by `::` (or by `'`, but that’s deprecated); all but the last are interpreted as names of packages, to locate the namespace in which to look up the final identifier (see the `Pack-ages` entry in the *perlmod* manpage for details). It’s possible to substitute for a simple identifier an expression that produces a reference to the value at runtime; this is described in more detail below, and in the *perlref* manpage.

There are also special variables whose names don’t follow these rules, so that they don’t accidentally collide with one of your normal variables. Strings that match parenthesized parts of a regular expression are saved under names containing only digits after the `$` (see the *perlop* manpage and the *perlre* manpage). In addition, several special variables that provide windows into the inner working of Perl have names containing punctuation characters (see the *perlvar* manpage).

Scalar values are always named with `$`, even when referring to a scalar that is part of an array. It works like the English word “the”. Thus we have:

```
$days           # the simple scalar value "days"
$days[28]      # the 29th element of array @days
$days{'Feb'}   # the 'Feb' value from hash %days
$#days         # the last index of array @days
```

but entire arrays or array slices are denoted by `@`, which works much like the word “these” or “those”:

```
@days          # ($days[0], $days[1], ... $days[n])
@days[3,4,5]   # same as @days[3..5]
@days{'a','c'} # same as ($days{'a'}, $days{'c'})
```

and entire hashes are denoted by `%`:

```
%days          # (key1, val1, key2, val2 ...)
```

In addition, subroutines are named with an initial `&`, though this is optional when it’s otherwise unambiguous (just as “do” is often redundant in English). Symbol table entries can be named with an initial `*`, but you don’t really care about that yet.

Every variable type has its own namespace. You can, without fear of conflict, use the same name for a scalar variable, an array, or a hash (or, for that matter, a filehandle, a subroutine name, or a label). This means that `$foo` and `@foo` are two different variables. It also means that `$foo[1]` is a part of `@foo`, not a part of `$foo`. This may seem a bit weird, but that's okay, because it is weird.

Because variable and array references always start with '\$', '@', or '%', the "reserved" words aren't in fact reserved with respect to variable names. (They ARE reserved with respect to labels and filehandles, however, which don't have an initial special character. You can't have a filehandle named "log", for instance. Hint: you could say `open(LOG, 'logfile')` rather than `open(log, 'logfile')`. Using uppercase filehandles also improves readability and protects you from conflict with future reserved words.) Case *IS* significant—"FOO", "Foo", and "foo" are all different names. Names that start with a letter or underscore may also contain digits and underscores.

It is possible to replace such an alphanumeric name with an expression that returns a reference to an object of that type. For a description of this, see the *perlref* manpage.

Names that start with a digit may contain only more digits. Names that do not start with a letter, underscore, or digit are limited to one character, e.g., `$%` or `$$`. (Most of these one character names have a predefined significance to Perl. For instance, `$$` is the current process id.)

### Context

The interpretation of operations and values in Perl sometimes depends on the requirements of the context around the operation or value. There are two major contexts: scalar and list. Certain operations return list values in contexts wanting a list, and scalar values otherwise. (If this is true of an operation it will be mentioned in the documentation for that operation.) In other words, Perl overloads certain operations based on whether the expected return value is singular or plural. (Some words in English work this way, like "fish" and "sheep".)

In a reciprocal fashion, an operation provides either a scalar or a list context to each of its arguments. For example, if you say

```
int( <STDIN> )
```

the integer operation provides a scalar context for the `<STDIN>` operator, which responds by reading one line from STDIN and passing it back to the integer operation, which will then find the integer value of that line and return that. If, on the other hand, you say

```
sort( <STDIN> )
```

then the sort operation provides a list context for `<STDIN>`, which will proceed to read every line available up to the end of file, and pass that list of lines back to the sort routine, which will then sort those lines and return them as a list to whatever the context of the sort was.

Assignment is a little bit special in that it uses its left argument to determine the context for the right argument. Assignment to a scalar evaluates the righthand side in a scalar context, while assignment to an array or array slice evaluates the righthand side in a list context. Assignment to a list also evaluates the righthand side in a list context.

User defined subroutines may choose to care whether they are being called in a scalar or list context, but most subroutines do not need to care, because scalars are automatically interpolated into lists. See the `wantarray` entry in the *perlfunc* manpage.

### Scalar values

All data in Perl is a scalar or an array of scalars or a hash of scalars. Scalar variables may contain various kinds of singular data, such as numbers, strings, and references. In general, conversion from one form to another is transparent. (A scalar may not contain multiple values, but may contain a reference to an array or hash containing multiple values.) Because of the automatic conversion of scalars, operations, and functions that return scalars don't need to care (and, in fact, can't care) whether the context is looking for a string or a number.

Scalars aren't necessarily one thing or another. There's no place to declare a scalar variable to be of type "string", or of type "number", or type "filehandle", or anything else. Perl is a contextually polymorphic language whose scalars can be strings, numbers, or references (which includes objects). While strings and numbers are considered pretty much the same thing for nearly all purposes, references are strongly-typed uncastable pointers with builtin reference-counting and destructor invocation.

A scalar value is interpreted as TRUE in the Boolean sense if it is not the null string or the number 0 (or its string equivalent, "0"). The Boolean context is just a special kind of scalar context.

There are actually two varieties of null scalars: defined and undefined. Undefined null scalars are returned when there is no real value for something, such as when there was an error, or at end of file, or when you refer to an uninitialized variable or element of an array. An undefined null scalar may become defined the first time you use it as if it were defined, but prior to that you can use the *defined()* operator to determine whether the value is defined or not.

To find out whether a given string is a valid nonzero number, it's usually enough to test it against both numeric 0 and also lexical "0" (although this will cause `-w` noises). That's because strings that aren't numbers count as 0, just as they do in **awk**:

```
if ($str == 0 && $str ne "0") {
    warn "That doesn't look like a number";
}
```

That's usually preferable because otherwise you won't treat IEEE notations like NaN or Infinity properly. At other times you might prefer to use the POSIX::strtod function or a regular expression to check whether data is numeric. See the *perlre* manpage for details on regular expressions.

```
warn "has nondigits"          if /\D/;
warn "not a natural number"  unless /^?\d+$/;           # rejects -3
warn "not an integer"       unless /^-?\d+$/;           # rejects +3
warn "not an integer"       unless /^[+-]?\d+$/;
warn "not a decimal number" unless /^-?\d+\.\d*$/;      # rejects .2
warn "not a decimal number" unless /^-?(?:\d+(?:\.\d*)?|\.\d+)$/;
warn "not a C float"
    unless /^([+-]?)(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?$/;
```

The length of an array is a scalar value. You may find the length of array `@days` by evaluating `$#days`, as in **cs**. (Actually, it's not the length of the array, it's the

subscript of the last element, because there is (ordinarily) a 0th element.) Assigning to  `$#days` changes the length of the array. Shortening an array by this method destroys intervening values. Lengthening an array that was previously shortened *NO LONGER* recovers the values that were in those elements. (It used to in Perl 4, but we had to break this to make sure destructors were called when expected.) You can also gain some miniscule measure of efficiency by pre-extending an array that is going to get big. (You can also extend an array by assigning to an element that is off the end of the array.) You can truncate an array down to nothing by assigning the null list `()` to it. The following are equivalent:

```
@whatever = ();
 $#whatever = -1;
```

If you evaluate a named array in a scalar context, it returns the length of the array. (Note that this is not true of lists, which return the last value, like the C comma operator, nor of built-in functions, which return whatever they feel like returning.) The following is always true:

```
scalar(@whatever) == $#whatever - $[ + 1;
```

Version 5 of Perl changed the semantics of `$[:`: files that don't set the value of `$[` no longer need to worry about whether another file changed its value. (In other words, use of `$[` is deprecated.) So in general you can assume that

```
scalar(@whatever) == $#whatever + 1;
```

Some programmers choose to use an explicit conversion so nothing's left to doubt:

```
$element_count = scalar(@whatever);
```

If you evaluate a hash in a scalar context, it returns a value that is true if and only if the hash contains any key/value pairs. (If there are any key/value pairs, the value returned is a string consisting of the number of used buckets and the number of allocated buckets, separated by a slash. This is pretty much useful only to find out whether Perl's (compiled in) hashing algorithm is performing poorly on your data set. For example, you stick 10,000 things in a hash, but evaluating `%HASH` in scalar context reveals "1/16", which means only one out of sixteen buckets has been touched, and presumably contains all 10,000 of your items. This isn't supposed to happen.)

You can preallocate space for a hash by assigning to the `keys()` function. This rounds up the allocated bucket to the next power of two:

```
keys(%users) = 1000; # allocate 1024 buckets
```

### Scalar value constructors

Numeric literals are specified in any of the customary floating point or integer formats:

```
12345
12345.67
.23E-10
0xffff # hex
0377 # octal
4_294_967_296 # underline for legibility
```

String literals are usually delimited by either single or double quotes. They work much like shell quotes: double-quoted string literals are subject to backslash and variable substitution; single-quoted strings are not (except for “\’” and “\\”). The usual Unix backslash rules apply for making characters such as newline, tab, etc., as well as some more exotic forms. See the section on *Quote and Quotelike Operators* in the *perlop* manpage for a list.

Octal or hex representations in string literals (e.g. ‘0xffff’) are not automatically converted to their integer representation. The *hex()* and *oct()* functions make these conversions for you. See the *hex* entry in the *perlfunc* manpage and the *oct* entry in the *perlfunc* manpage for more details.

You can also embed newlines directly in your strings, i.e., they can end on a different line than they begin. This is nice, but if you forget your trailing quote, the error will not be reported until Perl finds another line containing the quote character, which may be much further on in the script. Variable substitution inside strings is limited to scalar variables, arrays, and array slices. (In other words, names beginning with \$ or @, followed by an optional bracketed expression as a subscript.) The following code segment prints out “The price is \$100.”

```
$Price = '$100';    # not interpreted
print "The price is $Price.\n";    # interpreted
```

As in some shells, you can put curly brackets around the name to delimit it from following alphanumerics. In fact, an identifier within such curlies is forced to be a string, as is any single identifier within a hash subscript. Our earlier example,

```
$days{ 'Feb' }
```

can be written as

```
$days{Feb}
```

and the quotes will be assumed automatically. But anything more complicated in the subscript will be interpreted as an expression.

Note that a single-quoted string must be separated from a preceding word by a space, because single quote is a valid (though deprecated) character in a variable name (see the *Packages* entry in the *perlmod* manpage).

Three special literals are `__FILE__`, `__LINE__`, and `__PACKAGE__`, which represent the current filename, line number, and package name at that point in your program. They may be used only as separate tokens; they will not be interpolated into strings. If there is no current package (due to an empty `package ;` directive), `__PACKAGE__` is the undefined value.

The tokens `__END__` and `__DATA__` may be used to indicate the logical end of the script before the actual end of file. Any following text is ignored, but may be read via a DATA filehandle: `main::DATA` for `__END__`, or `PACKNAME::DATA` (where `PACKNAME` is the current package) for `__DATA__`. The two control characters `^D` and `^Z` are synonyms for `__END__` (or `__DATA__` in a module). See the *SelfLoader* manpage for more description of `__DATA__`, and an example of its use. Note that you cannot read from the DATA filehandle in a BEGIN block: the BEGIN block is executed as soon as it is seen (during compilation), at which point the corresponding `__DATA__` (or `__END__`) token has not yet been seen.

A word that has no other interpretation in the grammar will be treated as if it were a quoted string. These are known as “barewords”. As with filehandles and labels, a bareword that consists entirely of lowercase letters risks conflict with future reserved words, and if you use the `-w` switch, Perl will warn you about any such words. Some people may wish to outlaw barewords entirely. If you say

```
use strict 'subs';
```

then any bareword that would NOT be interpreted as a subroutine call produces a compile-time error instead. The restriction lasts to the end of the enclosing block. An inner block may countermand this by saying `no strict 'subs'`.

Array variables are interpolated into double-quoted strings by joining all the elements of the array with the delimiter specified in the `$"` variable (`$LIST_SEPARATOR` in English), space by default. The following are equivalent:

```
$temp = join("$", @ARGV);
system "echo $temp";

system "echo @ARGV";
```

Within search patterns (which also undergo double-quotish substitution) there is a bad ambiguity: Is `/$foo[bar]/` to be interpreted as `/${foo}[bar]/` (where `[bar]` is a character class for the regular expression) or as `/${foo}[bar]}/` (where `[bar]` is the subscript to array `@foo`)? If `@foo` doesn't otherwise exist, then it's obviously a character class. If `@foo` exists, Perl takes a good guess about `[bar]`, and is almost always right. If it does guess wrong, or if you're just plain paranoid, you can force the correct interpretation with curly brackets as above.

A line-oriented form of quoting is based on the shell “here-doc” syntax. Following a `<<` you specify a string to terminate the quoted material, and all lines following the current line down to the terminating string are the value of the item. The terminating string may be either an identifier (a word), or some quoted text. If quoted, the type of quotes you use determines the treatment of the text, just as in regular quoting. An unquoted identifier works like double quotes. There must be no space between the `<<` and the identifier. (If you put a space it will be treated as a null identifier, which is valid, and matches the first empty line.) The terminating string must appear by itself (unquoted and with no surrounding whitespace) on the terminating line.

```
print <<EOF;
The price is $Price.
EOF

print <<"EOF"; # same as above
The price is $Price.
EOF

print <<`EOC`; # execute commands
echo hi there
echo lo there
EOC
```

```

    print <<"foo", <<"bar"; # you can stack them
I said foo.
foo
I said bar.
bar

    myfunc(<<"THIS", 23, <<'THAT');
Here's a line
or two.
THIS
and here's another.
THAT

```

Just don't forget that you have to put a semicolon on the end to finish the statement, as Perl doesn't know you're not going to try to do this:

```

    print <<ABC
179231
ABC
    + 20;

```

### List value constructors

List values are denoted by separating individual values by commas (and enclosing the list in parentheses where precedence requires it):

```
(LIST)
```

In a context not requiring a list value, the value of the list literal is the value of the final element, as with the C comma operator. For example,

```
@foo = ('cc', '-E', $bar);
```

assigns the entire list value to array foo, but

```
$foo = ('cc', '-E', $bar);
```

assigns the value of variable bar to variable foo. Note that the value of an actual array in a scalar context is the length of the array; the following assigns the value 3 to \$foo:

```
@foo = ('cc', '-E', $bar);
$foo = @foo;           # $foo gets 3
```

You may have an optional comma before the closing parenthesis of a list literal, so that you can say:

```
@foo = (
    1,
    2,
    3,
);
```

LISTs do automatic interpolation of sublist. That is, when a LIST is evaluated, each element of the list is evaluated in a list context, and the resulting list value is

interpolated into LIST just as if each individual element were a member of LIST. Thus arrays and hashes lose their identity in a LIST—the list

```
(@foo, @bar, &SomeSub, %glarch)
```

contains all the elements of @foo followed by all the elements of @bar, followed by all the elements returned by the subroutine named SomeSub called in a list context, followed by the key/value pairs of %glarch. To make a list reference that does *NOT* interpolate, see the *perlref* manpage.

The null list is represented by (). Interpolating it in a list has no effect. Thus ((),(),()) is equivalent to (). Similarly, interpolating an array with no elements is the same as if no array had been interpolated at that point.

A list value may also be subscripted like a normal array. You must put the list in parentheses to avoid ambiguity. For example:

```
# Stat returns list value.
$time = (stat($file))[8];

# SYNTAX ERROR HERE.
$time = stat($file)[8]; # OOPS, FORGOT PARENTHESES

# Find a hex digit.
$hexdigit = ('a','b','c','d','e','f')[$digit-10];

# A "reverse comma operator".
return (pop(@foo),pop(@foo))[0];
```

You may assign to undef in a list. This is useful for throwing away some of the return values of a function:

```
($dev, $ino, undef, undef, $uid, $gid) = stat($file);
```

Lists may be assigned to if and only if each element of the list is legal to assign to:

```
($a, $b, $c) = (1, 2, 3);

($map{'red'}, $map{'blue'}, $map{'green'}) = (0x00f, 0x0f0, 0xf00);
```

Array assignment in a scalar context returns the number of elements produced by the expression on the right side of the assignment:

```
$x = (($foo,$bar) = (3,2,1)); # set $x to 3, not 2
$x = (($foo,$bar) = f()); # set $x to f()'s return count
```

This is very handy when you want to do a list assignment in a Boolean context, because most list functions return a null list when finished, which when assigned produces a 0, which is interpreted as FALSE.

The final element may be an array or a hash:

```
($a, $b, @rest) = split;
my($a, $b, %rest) = @_;
```

You can actually put an array or hash anywhere in the list, but the first one in the list will soak up all the values, and anything after it will get a null value. This may be

useful in a *local()* or *my()*.

A hash literal contains pairs of values to be interpreted as a key and a value:

```
# same as map assignment above
%map = ( 'red', 0x00f, 'blue', 0x0f0, 'green', 0xf00 );
```

While literal lists and named arrays are usually interchangeable, that's not the case for hashes. Just because you can subscript a list value like a normal array does not mean that you can subscript a list value as a hash. Likewise, hashes included as parts of other lists (including parameters lists and return lists from functions) always flatten out into key/value pairs. That's why it's good to use references sometimes.

It is often more readable to use the `=>` operator between key/value pairs. The `=>` operator is mostly just a more visually distinctive synonym for a comma, but it also arranges for its left-hand operand to be interpreted as a string—if it's a bareword that would be a legal identifier. This makes it nice for initializing hashes:

```
%map = (
    red    => 0x00f,
    blue   => 0x0f0,
    green  => 0xf00,
);
```

or for initializing hash references to be used as records:

```
$rec = {
    witch => 'Mable the Merciless',
    cat   => 'Fluffy the Ferocious',
    date  => '10/31/1776',
};
```

or for using call-by-named-parameter to complicated functions:

```
$field = $query->radio_group(
    name      => 'group_name',
    values    => ['eenie', 'meenie', 'minie'],
    default   => 'meenie',
    linebreak => 'true',
    labels    => \%labels
);
```

Note that just because a hash is initialized in that order doesn't mean that it comes out in that order. See the `sort` entry in the *perlfunc* manpage for examples of how to arrange for an output ordering.

### Typeglobs and Filehandles

Perl uses an internal type called a *typeglob* to hold an entire symbol table entry. The type prefix of a typeglob is a `*`, because it represents all types. This used to be the preferred way to pass arrays and hashes by reference into a function, but now that we have real references, this is seldom needed.

The main use of typeglobs in modern Perl is create symbol table aliases. This assignment:

```
*this = *that;
```

makes `$this` an alias for `$that`, `@this` an alias for `@that`, `%this` an alias for `%that`, `&this` an alias for `&that`, etc. Much safer is to use a reference. This:

```
local *Here::blue = \$There::green;
```

temporarily makes `$Here::blue` an alias for `$There::green`, but doesn't make `@Here::blue` an alias for `@There::green`, or `%Here::blue` an alias for `%There::green`, etc. See the section on *Symbol Tables* in the *perlmod* manpage for more examples of this. Strange though this may seem, this is the basis for the whole module import/export system.

Another use for typeglobs is to pass filehandles into a function or to create new filehandles. If you need to use a typeglob to save away a filehandle, do it this way:

```
$fh = *STDOUT;
```

or perhaps as a real reference, like this:

```
$fh = \*STDOUT;
```

See the *perlsub* manpage for examples of using these as indirect filehandles in functions.

Typeglobs are also a way to create a local filehandle using the *local()* operator. These last until their block is exited, but may be passed back. For example:

```
sub newopen {
    my $path = shift;
    local *FH; # not my!
    open (FH, $path) or return undef;
    return *FH;
}
$fh = newopen('/etc/passwd');
```

Now that we have the `*foo{THING}` notation, typeglobs aren't used as much for filehandle manipulations, although they're still needed to pass brand new file and directory handles into or out of functions. That's because `*HANDLE{IO}` only works if `HANDLE` has already been used as a handle. In other words, `*FH` can be used to create new symbol table entries, but `*foo{THING}` cannot.

Another way to create anonymous filehandles is with the `IO::Handle` module and its ilk. These modules have the advantage of not hiding different types of the same name during the *local()*. See the bottom of the `open()` entry in the *perlfunc* manpage for an example.

See the *perlref* manpage, the *perlsub* manpage, and the section on *Symbol Tables* in the *perlmod* manpage for more discussion on typeglobs and the `*foo{THING}` syntax.

