NAME

perldata - Perl data types

DESCRIPTION

Variable names

Perl has three built-in data types: scalars, arrays of scalars, and associative arrays of scalars, known as "hashes". A scalar is a single string (of any size, limited only by the available memory), number, or a reference to something (which will be discussed in *perlref*). Normal arrays are ordered lists of scalars indexed by number, starting with 0. Hashes are unordered collections of scalar values indexed by their associated string key.

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. Usually this name is 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 the slightly archaic `'`); all but the last are interpreted as names of packages, to locate the namespace in which to look up the final identifier (see *"Packages" in perlmod* for details). For a more in-depth discussion on identifiers, see *Identifier parsing*. 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 *perlref*.

Perl also has its own built-in variables whose names don't follow these rules. They have strange names so 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 *perlop* and *perlre*). In addition, several special variables that provide windows into the inner working of Perl have names containing punctuation characters. These are documented in *perlvar*.

Scalar values are always named with '$', even when referring to a scalar that is part of an array or a hash. The '$' symbol works semantically like the English word "the" in that it indicates a single value is expected.

```
$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
```

Entire arrays (and slices of arrays and hashes) are denoted by '@', which works much as the word "these" or "those" does in English, in that it indicates multiple values are expected.

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

Entire hashes are denoted by '%':

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

In addition, subroutines are named with an initial '&', though this is optional when unambiguous, just as the word "do" is often redundant in English. Symbol table entries can be named with an initial '*', but you don't really care about that yet (if ever :-).

Every variable type has its own namespace, as do several non-variable identifiers. This means that you can, without fear of conflict, use the same name for a scalar variable, an array, or a hash--or, for that matter, for a filehandle, a directory handle, a subroutine name, a format 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 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 the appropriate type. For a description of this, see *perlref*.

Names that start with a digit may contain only more digits. Names that do not start with a letter, underscore, digit or a caret 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. And all such names are reserved for Perl's possible use.)

Identifier parsing

Up until Perl 5.18, the actual rules of what a valid identifier was were a bit fuzzy. However, in general, anything defined here should work on previous versions of Perl, while the opposite -- edge cases that work in previous versions, but aren't defined here -- probably won't work on newer versions. As an important side note, please note that the following only applies to bareword identifiers as found in Perl source code, not identifiers introduced through symbolic references, which have much fewer restrictions. If working under the effect of the `use utf8;` pragma, the following rules apply:

```
/ (?[ ( \p{Word} & \p{XID_Start} ) + [_] ])
  (?[ ( \p{Word} & \p{XID_Continue} ) ]) *    /x
```

That is, a "start" character followed by any number of "continue" characters. Perl requires every character in an identifier to also match `\w` (this prevents some problematic cases); and Perl additionally accepts identfier names beginning with an underscore.

If not under `use utf8`, the source is treated as ASCII + 128 extra generic characters, and identifiers should match

```
/ (?aa) (?!\d) \w+ /x
```

That is, any word character in the ASCII range, as long as the first character is not a digit.

There are two package separators in Perl: A double colon (`::`) and a single quote (`'`). Normal identifiers can start or end with a double colon, and can contain several parts delimited by double colons. Single quotes have similar rules, but with the exception that they are not legal at the end of an identifier: That is, `$'foo` and `$foo'bar` are legal, but `$foo'bar'` is not.

Additionally, if the identifier is preceded by a sigil -- that is, if the identifier is part of a variable name -- it may optionally be enclosed in braces.

While you can mix double colons with singles quotes, the quotes must come after the colons: `$::::'foo` and `$foo::'bar` are legal, but `$::'::foo` and `$foo'::bar` are not.

Put together, a grammar to match a basic identifier becomes

```
/
 (?(DEFINE)
     (?<variable>
         (?&sigil)
         (?:
                 (?&normal_identifier)
             |   \{ \s* (?&normal_identifier) \s* \}
         )
```

```
        )
        (?<normal_identifier>
            (?: :: )* '?
             (?&basic_identifier)
             (?: (?= (?: :: )+ '? | (?: :: )* ' ) (?&normal_identifier) )?
            (?: :: )*
        )
        (?<basic_identifier>
          # is use utf8 on?
            (?(?{ (caller(0))[8] & $utf8::hint_bits })
                (?&Perl_XIDS) (?&Perl_XIDC)*
              | (?aa) (?!\d) \w+
            )
        )
        (?<sigil> [&*\$\@\%])
        (?<Perl_XIDS> (?[ ( \p{Word} & \p{XID_Start} ) + [_] ]) )
        (?<Perl_XIDC> (?[ \p{Word} & \p{XID_Continue} ]) )
    )
   /x
```

Meanwhile, special identifiers don't follow the above rules; For the most part, all of the identifiers in this category have a special meaning given by Perl. Because they have special parsing rules, these generally can't be fully-qualified. They come in six forms (but don't use forms 5 and 6):

1. A sigil, followed solely by digits matching `\p{POSIX_Digit}`, like `$0`, `$1`, or `$10000`.

2. A sigil followed by a single character matching the `\p{POSIX_Punct}` property, like `$!` or `%+`, except the character `"{"` doesn't work.

3. A sigil, followed by a caret and any one of the characters `[][A-Z^_?\]`, like `$^V` or `$^]`.

4. Similar to the above, a sigil, followed by bareword text in braces, where the first character is a caret. The next character is any one of the characters `[][A-Z^_?\]`, followed by ASCII word characters. An example is `${^GLOBAL_PHASE}`.

5. A sigil, followed by any single character in the range `[\xA1-\xAC\xAE-\xFF]` when not under `"use utf8"`. (Under `"use utf8"`, the normal identifier rules given earlier in this section apply.) Use of non-graphic characters (the C1 controls, the NO-BREAK SPACE, and the SOFT HYPHEN) has been disallowed since v5.26.0. The use of the other characters is unwise, as these are all reserved to have special meaning to Perl, and none of them currently do have special meaning, though this could change without notice.

   Note that an implication of this form is that there are identifiers only legal under `"use utf8"`, and vice-versa, for example the identifier `$état` is legal under `"use utf8"`, but is otherwise considered to be the single character variable `$é` followed by the bareword `"tat"`, the combination of which is a syntax error.

6. This is a combination of the previous two forms. It is valid only when not under `"use utf8"` (normal identifier rules apply when under `"use utf8"`). The form is a sigil, followed by text in braces, where the first character is any one of the characters in the range `[\x80-\xFF]` followed by ASCII word characters up to the trailing brace.

   The same caveats as the previous form apply: The non-graphic characters are no longer allowed with "use utf8", it is unwise to use this form at all, and utf8ness makes a big difference.

Prior to Perl v5.24, non-graphical ASCII control characters were also allowed in some situations; this had been deprecated since v5.20.

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: list and scalar. 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 scalar context for the <> 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 list context for <>, 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 right-hand side in scalar context, while assignment to an array or hash evaluates the righthand side in list context. Assignment to a list (or slice, which is just a list anyway) also evaluates the right-hand side in list context.

When you use the `use warnings` pragma or Perl's **-w** command-line option, you may see warnings about useless uses of constants or functions in "void context". Void context just means the value has been discarded, such as a statement containing only `"fred";` or `getpwuid(0);`. It still counts as scalar context for functions that care whether or not they're being called in list context.

User-defined subroutines may choose to care whether they are being called in a void, scalar, or list context. Most subroutines do not need to bother, though. That's because both scalars and lists are automatically interpolated into lists. See *"wantarray" in perlfunc* for how you would dynamically discern your function's calling context.

Scalar values

All data in Perl is a scalar, an array of scalars, or a hash of scalars. A scalar may contain one single value in any of three different flavors: a number, a string, or a reference. In general, conversion from one form to another is transparent. Although a scalar may not directly hold multiple values, it may contain a reference to an array or hash which in turn contains multiple values.

Scalars aren't necessarily one thing or another. There's no place to declare a scalar variable to be of type "string", type "number", type "reference", or anything else. Because of the automatic conversion of scalars, operations that return scalars don't need to care (and in fact, cannot care) whether their caller is looking for a string, a number, or a reference. Perl is a contextually polymorphic language whose scalars can be strings, numbers, or references (which includes objects). Although 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 FALSE in the Boolean sense if it is undefined, the null string or the number 0 (or its string equivalent, "0"), and TRUE if it is anything else. The Boolean context is just a special kind of scalar context where no conversion to a string or a number is ever performed.

There are actually two varieties of null strings (sometimes referred to as "empty" strings), a defined one and an undefined one. The defined version is just a string of length zero, such as `""`. The

undefined version is the value that indicates that 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 or hash. Although in early versions of Perl, an undefined scalar could become defined when first used in a place expecting a defined value, this no longer happens except for rare cases of autovivification as explained in *perlref*. You can use the defined() operator to determine whether a scalar value is defined (this has no meaning on arrays or hashes), and the undef() operator to produce an undefined value.

To find out whether a given string is a valid non-zero number, it's sometimes enough to test it against both numeric 0 and also lexical "0" (although this will cause noises if warnings are on). 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 method may be best because otherwise you won't treat IEEE notations like `NaN` or `Infinity` properly. At other times, you might prefer to determine whether string data can be used numerically by calling the POSIX::strtod() function or by inspecting your string with a regular expression (as documented in *perlre*).

```
    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 `$#days`, as in **csh**. However, this isn't the length of the array; it's the subscript of the last element, which is a different value since there is ordinarily a 0th element. Assigning to `$#days` actually changes the length of the array. Shortening an array this way destroys intervening values. Lengthening an array that was previously shortened does not recover values that were in those elements.

You can also gain some minuscule 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 an array in 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;
```

Some programmers choose to use an explicit conversion so as to leave nothing to doubt:

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

If you evaluate a hash in scalar context, it returns false if the hash is empty. If there are any key/value pairs, it returns true. A more precise definition is version dependent.

Prior to Perl 5.25 the value returned was 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 internal 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.

As of Perl 5.25 the return was changed to be the count of keys in the hash. If you need access to the old behavior you can use `Hash::Util::bucket_ratio()` instead.

If a tied hash is evaluated in scalar context, the SCALAR method is called (with a fallback to FIRSTKEY).

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

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

Scalar value constructors

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

```
12345
12345.67
.23E-10             # a very small number
3.14_15_92          # a very important number
4_294_967_296       # underscore for legibility
0xff                # hex
0xdead_beef         # more hex
0377                # octal (only numbers, begins with 0)
0b011011            # binary
0x1.999ap-4         # hexadecimal floating point (the 'p' is required)
```

You are allowed to use underscores (underbars) in numeric literals between digits for legibility (but not multiple underscores in a row: `23__500` is not legal; `23_500` is). You could, for example, group binary digits by threes (as for a Unix-style mode argument such as 0b110_100_100) or by fours (to represent nibbles, as in 0b1010_0110) or in other groups.

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

Hexadecimal, octal, or binary, representations in string literals (e.g. '0xff') are not automatically converted to their integer representation. The hex() and oct() functions make these conversions for you. See *"hex" in perlfunc* and *"oct" in perlfunc* for more details.

Hexadecimal floating point can start just like a hexadecimal literal, and it can be followed by an optional fractional hexadecimal part, but it must be followed by `p`, an optional sign, and a power of two. The format is useful for accurately presenting floating point values, avoiding conversions to or from decimal floating point, and therefore avoiding possible loss in precision. Notice that while most current platforms use the 64-bit IEEE 754 floating point, not all do. Another potential source of (low-order) differences are the floating point rounding modes, which can differ between CPUs, operating systems, and compilers, and which Perl doesn't control.

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 or hash 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 interpolated
print "The price is $Price.\n"; # interpolated
```

There is no double interpolation in Perl, so the `$100` is left as is.

By default floating point numbers substituted inside strings use the dot (".") as the decimal separator. If `use locale` is in effect, and POSIX::setlocale() has been called, the character used for the decimal separator is affected by the LC_NUMERIC locale. See *perllocale* and *POSIX*.

As in some shells, you can enclose the variable name in braces to disambiguate it from following alphanumerics (and underscores). You must also do this when interpolating a variable into a string to separate the variable name from a following double-colon or an apostrophe, since these would be otherwise treated as a package separator:

```
$who = "Larry";
print PASSWD "${who}::0:0:Superuser:/:/bin/perl\n";
print "We use ${who}speak when ${who}'s here.\n";
```

Without the braces, Perl would have looked for a $whospeak, a `$who::0`, and a `$who's` variable. The last two would be the $0 and the $s variables in the (presumably) non-existent package `who`.

In fact, a simple identifier within such curlies is forced to be a string, and likewise within a hash subscript. Neither need quoting. 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. This means for example that `$version{2.0}++` is equivalent to `$version{2}++`, not to `$version{'2.0'}++`.

Special floating point: infinity (Inf) and not-a-number (NaN)

Floating point values include the special values `Inf` and `NaN`, for infinity and not-a-number. The infinity can be also negative.

The infinity is the result of certain math operations that overflow the floating point range, like 9**9**9. The not-a-number is the result when the result is undefined or unrepresentable. Though note that you cannot get `NaN` from some common "undefined" or "out-of-range" operations like dividing by zero, or square root of a negative number, since Perl generates fatal errors for those.

The infinity and not-a-number have their own special arithmetic rules. The general rule is that they are "contagious": `Inf` plus one is `Inf`, and `NaN` plus one is `NaN`. Where things get interesting is when you combine infinities and not-a-numbers: `Inf` minus `Inf` and `Inf` divided by `Inf` are `NaN` (while `Inf` plus `Inf` is `Inf` and `Inf` times `Inf` is `Inf`). `NaN` is also curious in that it does not equal any number, *including* itself: `NaN` != `NaN`.

Perl doesn't understand `Inf` and `NaN` as numeric literals, but you can have them as strings, and Perl will convert them as needed: "Inf" + 1. (You can, however, import them from the POSIX extension; `use POSIX qw(Inf NaN);` and then use them as literals.)

Note that on input (string to number) Perl accepts `Inf` and `NaN` in many forms. Case is ignored, and the Win32-specific forms like `1.#INF` are understood, but on output the values are normalized to `Inf` and `NaN`.

Version Strings

A literal of the form `v1.20.300.4000` is parsed as a string composed of characters with the specified ordinals. This form, known as v-strings, provides an alternative, more readable way to construct strings, rather than use the somewhat less readable interpolation form

"\x{1}\x{14}\x{12c}\x{fa0}". This is useful for representing Unicode strings, and for comparing version "numbers" using the string comparison operators, cmp, gt, lt etc. If there are two or more dots in the literal, the leading v may be omitted.

```
print v9786;              # prints SMILEY, "\x{263a}"
print v102.111.111;       # prints "foo"
print 102.111.111;        # same
```

Such literals are accepted by both require and use for doing a version check. Note that using the v-strings for IPv4 addresses is not portable unless you also use the inet_aton()/inet_ntoa() routines of the Socket package.

Note that since Perl 5.8.1 the single-number v-strings (like v65) are not v-strings before the => operator (which is usually used to separate a hash key from a hash value); instead they are interpreted as literal strings ('v65'). They were v-strings from Perl 5.6.0 to Perl 5.8.0, but that caused more confusion and breakage than good. Multi-number v-strings like v65.66 and 65.66.67 continue to be v-strings always.

Special Literals

The special literals __FILE__, __LINE__, and __PACKAGE__ represent the current filename, line number, and package name at that point in your program. __SUB__ gives a reference to the current subroutine. 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. (But the empty package; is no longer supported, as of version 5.10.) Outside of a subroutine, __SUB__ is the undefined value. __SUB__ is only available in 5.16 or higher, and only with a use v5.16 or use feature "current_sub" declaration.

The two control characters ^D and ^Z, and 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.

Text after __DATA__ may be read via the filehandle PACKNAME::DATA, where PACKNAME is the package that was current when the __DATA__ token was encountered. The filehandle is left open pointing to the line after __DATA__. The program should close DATA when it is done reading from it. (Leaving it open leaks filehandles if the module is reloaded for any reason, so it's a safer practice to close it.) For compatibility with older scripts written before __DATA__ was introduced, __END__ behaves like __DATA__ in the top level script (but not in files loaded with require or do) and leaves the remaining contents of the file accessible via main::DATA.

See *SelfLoader* 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.

Barewords

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 use warnings pragma or the **-w** switch, Perl will warn you about any such words. Perl limits barewords (like identifiers) to about 250 characters. Future versions of Perl are likely to eliminate these arbitrary limitations.

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 Interpolation

Arrays and slices are interpolated into double-quoted strings by joining the elements with the delimiter specified in the `$"` variable (`$LIST_SEPARATOR` if "use English;" is specified), 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 an unfortunate 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 braces as above.

If you're looking for the information on how to use here-documents, which used to be here, that's been moved to *"Quote and Quote-like Operators" in perlop*.

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 what appears to be a list literal is simply 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 the scalar variable $foo. Note that the value of an actual array in 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,
);
```

To use a here-document to assign an array, one line per element, you might use an approach like this:

```
@sauces = <<End_Lines =~ m/(\S.*\S)/g;
    normal tomato
    spicy tomato
    green chile
```

```
        pesto
        white wine
    End_Lines
```

LISTs do automatic interpolation of sublists. That is, when a LIST is evaluated, each element of the list is evaluated in 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 list context, followed by the key/value pairs of %glarch. To make a list reference that does *NOT* interpolate, see *perlref*.

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.

This interpolation combines with the facts that the opening and closing parentheses are optional (except when necessary for precedence) and lists may end with an optional comma to mean that multiple commas within lists are legal syntax. The list `1,3` is a concatenation of two lists, `1,` and `3`, the first of which ends with that optional comma. `1,,3` is `(1,),(3)` is `1,3` (And similarly for `1,,,3` is `(1,),(,),3` is `1,3` and so on.) Not that we'd advise you to use this obfuscation.

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];
```

Lists may be assigned to only when each element of the list is itself legal to assign to:

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

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

An exception to this is that 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);
```

As of Perl 5.22, you can also use `(undef)x2` instead of `undef, undef`. (You can also do `($x) x 2`, which is less useful, because it assigns to the same variable twice, clobbering the first value assigned.)

List assignment in 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 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.

It's also the source of a useful idiom for executing a function or performing an operation in list context and then counting the number of return values, by assigning to an empty list and then using that assignment in scalar context. For example, this code:

```
$count = () = $string =~ /\d+/g;
```

will place into $count the number of digit groups found in $string. This happens because the pattern match is in list context (since it is being assigned to the empty list), and will therefore return a list of all matching parts of the string. The list assignment in scalar context will translate that into the number of elements (here, the number of times the pattern matched) and assign that to $count. Note that simply using

```
$count = $string =~ /\d+/g;
```

would not have worked, since a pattern match in scalar context will only return true or false, rather than a count of matches.

The final element of a list assignment 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 become undefined. This may be useful in a my() or local().

A hash can be initialized using a literal list holding pairs of items 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 often 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 simple identifier. => doesn't quote compound identifiers, that contain double colons. 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 *"sort" in perlfunc* for examples of how to arrange for an output ordering.

If a key appears more than once in the initializer list of a hash, the last occurrence wins:

```
    %circle = (
                  center => [5, 10],
                  center => [27, 9],
                  radius => 100,
                  color => [0xDF, 0xFF, 0x00],
                  radius => 54,
    );

    # same as
    %circle = (
                  center => [27, 9],
                  color => [0xDF, 0xFF, 0x00],
                  radius => 54,
    );
```

This can be used to provide overridable configuration defaults:

```
    # values in %args take priority over %config_defaults
    %config = (%config_defaults, %args);
```

Subscripts

An array can be accessed one scalar at a time by specifying a dollar sign ($), then the name of the array (without the leading @), then the subscript inside square brackets. For example:

```
    @myarray = (5, 50, 500, 5000);
    print "The Third Element is", $myarray[2], "\n";
```

The array indices start with 0. A negative subscript retrieves its value from the end. In our example, `$myarray[-1]` would have been 5000, and `$myarray[-2]` would have been 500.

Hash subscripts are similar, only instead of square brackets curly brackets are used. For example:

```
    %scientists =
    (
        "Newton" => "Isaac",
```

```
    "Einstein" => "Albert",
    "Darwin" => "Charles",
    "Feynman" => "Richard",
);

print "Darwin's First Name is ", $scientists{"Darwin"}, "\n";
```

You can also subscript a list to get a single element from it:

```
$dir = (getpwnam("daemon"))[7];
```

Multi-dimensional array emulation

Multidimensional arrays may be emulated by subscripting a hash with a list. The elements of the list are joined with the subscript separator (see *"$;" in perlvar*).

```
$foo{$a,$b,$c}
```

is equivalent to

```
$foo{join($;, $a, $b, $c)}
```

The default subscript separator is "\034", the same as SUBSEP in **awk**.

Slices

A slice accesses several elements of a list, an array, or a hash simultaneously using a list of subscripts. It's more convenient than writing out the individual elements as a list of separate scalar values.

```
($him, $her)   = @folks[0,-1];              # array slice
@them          = @folks[0 .. 3];            # array slice
($who, $home)  = @ENV{"USER", "HOME"};      # hash slice
($uid, $dir)   = (getpwnam("daemon"))[2,7]; # list slice
```

Since you can assign to a list of variables, you can also assign to an array or hash slice.

```
@days[3..5]    = qw/Wed Thu Fri/;
@colors{'red','blue','green'}
               = (0xff0000, 0x0000ff, 0x00ff00);
@folks[0, -1]  = @folks[-1, 0];
```

The previous assignments are exactly equivalent to

```
($days[3], $days[4], $days[5]) = qw/Wed Thu Fri/;
($colors{'red'}, $colors{'blue'}, $colors{'green'})
               = (0xff0000, 0x0000ff, 0x00ff00);
($folks[0], $folks[-1]) = ($folks[-1], $folks[0]);
```

Since changing a slice changes the original array or hash that it's slicing, a `foreach` construct will alter some--or even all--of the values of the array or hash.

```
foreach (@array[ 4 .. 10 ]) { s/peter/paul/ }

foreach (@hash{qw[key1 key2]}) {
    s/^\s+//;          # trim leading whitespace
    s/\s+$//;          # trim trailing whitespace
```

```
    s/(\w+)/\u\L$1/g;   # "titlecase" words
}
```

As a special exception, when you slice a list (but not an array or a hash), if the list evaluates to empty, then taking a slice of that empty list will always yield the empty list in turn. Thus:

```
@a = ()[0,1];          # @a has no elements
@b = (@a)[0,1];        # @b has no elements
@c = (sub{}->())[0,1]; # @c has no elements
@d = ('a','b')[0,1];   # @d has two elements
@e = (@d)[0,1,8,9];    # @e has four elements
@f = (@d)[8,9];        # @f has two elements
```

This makes it easy to write loops that terminate when a null list is returned:

```
while ( ($home, $user) = (getpwent)[7,0] ) {
    printf "%-8s %s\n", $user, $home;
}
```

As noted earlier in this document, the scalar sense of list assignment is the number of elements on the right-hand side of the assignment. The null list contains no elements, so when the password file is exhausted, the result is 0, not 2.

Slices in scalar context return the last item of the slice.

```
@a = qw/first second third/;
%h = (first => 'A', second => 'B');
$t = @a[0, 1];                  # $t is now 'second'
$u = @h{'first', 'second'};     # $u is now 'B'
```

If you're confused about why you use an '@' there on a hash slice instead of a '%', think of it like this. The type of bracket (square or curly) governs whether it's an array or a hash being looked at. On the other hand, the leading symbol ('$' or '@') on the array or hash indicates whether you are getting back a singular value (a scalar) or a plural one (a list).

Key/Value Hash Slices

Starting in Perl 5.20, a hash slice operation with the % symbol is a variant of slice operation returning a list of key/value pairs rather than just values:

```
%h = (blonk => 2, foo => 3, squink => 5, bar => 8);
%subset = %h{'foo', 'bar'}; # key/value hash slice
# %subset is now (foo => 3, bar => 8)
```

However, the result of such a slice cannot be localized, deleted or used in assignment. These are otherwise very much consistent with hash slices using the @ symbol.

Index/Value Array Slices

Similar to key/value hash slices (and also introduced in Perl 5.20), the % array slice syntax returns a list of index/value pairs:

```
@a = "a".."z";
@list = %a[3,4,6];
# @list is now (3, "d", 4, "e", 6, "g")
```

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 *"Symbol Tables" in perlmod* 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 *perlsub* 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` must be used to create new symbol table entries; `*foo{THING}` cannot. When in doubt, use `*FH`.

All functions that are capable of creating filehandles (open(), opendir(), pipe(), socketpair(), sysopen(), socket(), and accept()) automatically create an anonymous filehandle if the handle passed to them is an uninitialized scalar variable. This allows the constructs such as `open(my $fh, ...)` and `open(local $fh,...)` to be used to create filehandles that will conveniently be closed automatically when the scope ends, provided there are no other references to them. This largely eliminates the need for typeglobs when opening filehandles that must be passed around, as in the following example:

```
sub myopen {
    open my $fh, "@_"
```

```
            or die "Can't open '@_': $!";
        return $fh;
    }

    {
        my $f = myopen("</etc/motd");
        print <$f>;
        # $f implicitly closed here
    }
```

Note that if an initialized scalar variable is used instead the result is different: `my $fh='zzz'; open($fh, ...)` is equivalent to `open( *{'zzz'}, ...)`. `use strict 'refs'` forbids such practice.

Another way to create anonymous filehandles is with the Symbol module or 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 *"open" in perlfunc* for an example.

SEE ALSO

See *perlvar* for a description of Perl's built-in variables and a discussion of legal variable names. See *perlref*, *perlsub*, and *"Symbol Tables" in perlmod* for more discussion on typeglobs and the `*foo{THING}` syntax.