The Perl language is a combination of several individual pieces. Although spoken languages use nuance and tone of voice and intuition to communicate across gaps in knowledge and understanding, computers and source code require precision. You can write effective Perl code without knowing every detail of every language feature, but you must understand how they work together to write Perl code well.

Names

Names (or *identifiers*) are everywhere in Perl programs: you choose them for variables, functions, packages, classes, and even filehandles. Valid Perl names all begin with a letter or an underscore and may optionally include any combination of letters, numbers, and underscores. When the `utf8` pragma (*unicode*) is in effect, you may use any UTF-8 word characters in identifiers. These are valid Perl identifiers:

```
    my $name;
    my @_private_names;
    my %Names_to_Addresses;
    sub anAwkwardName3;

    # with use utf8; enabled
    package Ingy::DE<ouml>t::Net;
```

These are invalid Perl identifiers:

```
    my $invalid name; # space is invalid
    my @3;            # cannot start with number
    my %~flags;       # symbols invalid in name

    package a-lisp-style-name;
```

Names exist primarily for your benefit as a programmer. These rules apply only to literal names which appear in your source code, such as `sub fetch_pie` or `my $waffleiron`.

Only Perl's parser enforces the rules about identifier names. You may also refer to entities with names generated at runtime or provided as input to a program. These *symbolic lookups* provide flexibility at the expense of safety. Invoking functions or methods indirectly or looking up symbols in a namespace lets you bypass Perl's parser. Symbolic lookups can produce confusing code. As Mark Jason Dominus recommendshttp://perl.plover.com/varvarname.html, prefer a hash (*hashes*) or nested data structure (*nested_data_structures*) over variables named, for example, `$recipe1`, `$recipe2`, and so on.

Variable Names and Sigils

Variable names always have a leading *sigil* (a symbol) which indicates the type of the variable's value. *Scalar variables* (*scalars*) use the dollar sign (`$`). *Array variables* (*arrays*) use the at sign (`@`). *Hash variables* (*hashes*) use the percent sign (`%`):

```
    my $scalar;
    my @array;
    my %hash;
```

Sigils separate variables into different namespaces. It's possible--though confusing--to declare multiple variables of the same name with different types:

```
    my ($bad_name, @bad_name, %bad_name);
```

Perl won't get confused, though humans will.

The sigil of a variable changes depending on its use; this change is called *variant sigils*. As context determines how many items you expect from an operation or what type of data you expect to get, so the sigil governs how you manipulate the data of a variable. For example, use the scalar sigil ($) to access a single element of an array or a hash:

```
my $hash_element  = $hash{ $key };
my $array_element = $array[ $index ]

$hash{ $key }     = 'value';
$array[ $index ]  = 'item';
```

The parallel with amount context is important. Using a scalar element of an aggregate as an *lvalue* (the target of an assignment; on the *l*eft side of the = character) imposes scalar context (*context_philosophy*) on the *rvalue* (the value assigned; on the *r*ight side of the = character).

Similarly, accessing multiple elements of a hash or an array--an operation known as *slicing*--uses the at symbol (@) and imposes list context--even if the list itself has zero or one elements:

```
my @hash_elements  = @hash{ @keys };
my @array_elements = @array[ @indexes ];

my %hash;
@hash{ @keys }     = @values;
```

Given Perl's variant sigils, the most reliable way to determine the type of a variable--scalar, array, or hash--is to observe the operations performed on it. Arrays support indexed access through square brackets. Hashes support keyed access through curly brackets. Scalars have neither.

Namespaces

Perl allows you to collect similar functions and variables into their own unique named spaces--*namespaces* (*packages*). A namespace is a collection of symbols grouped under a globally unique name. Perl allows multi-level namespaces, with names joined by double colons (`::`). `DessertShop::IceCream` refers to a logical collection of related variables and functions, such as `scoop()` and `pour_hot_fudge()`.

Within a namespace, you may use the short name of its members. Outside of the namespace, you must refer to a member by its *fully-qualified name*. Within `DessertShop::IceCream`, `add_sprinkles()` refers to the same function as does `DessertShop::IceCream::add_sprinkles()` outside of the namespace.

Standard identifier rules apply to package names. By convention, the Perl core reserves lowercase package names for core pragmas (*pragmas*), such as `strict` and `warnings`. User-defined packages all start with uppercase letters. This is a policy enforced primarily by community guidelines.

All namespaces in Perl are globally visible. When Perl looks up a symbol in `DessertShop::IceCream::Freezer`, it looks in the `main::` symbol table for a symbol representing the `DessertShop::` namespace, in that namespace for the `IceCream::` namespace, and so on. Yet `Freezer::` is visible from outside of the `IceCream::` namespace. The nesting of the former within the latter is only a storage mechanism; it implies nothing about relationships between parent and child or sibling packages.

Only you as a programmer can make *logical* relationships between entities obvious--by choosing good names and organizing them well.

Variables

A *variable* in Perl is a storage location for a value (*values*). While a trivial program may manipulate values directly, most programs work with variables. Think of this like algebra: you manipulate symbols to describe formulas. It's easier to explain the Pythagorean theorem in terms of the variables a, b, and c than by intuiting its principle by producing a long list of valid values.

Variable Scopes

Your ability to access a variable within your program depends on the variable's scope (*scope*). Most variables in modern Perl programs have a lexical scope (*lexical_scope*) governed by the syntax of the program as written. Most lexical scopes are either the contents of blocks delimited by curly braces (`{` and `}`) or entire files. *Files* themselves provide their own lexical scopes, such that a `package` declaration on its own does not create a new scope:

```
package Store::Toy;

my $discount = 0.10;

package Store::Music;

# $discount still visible
say "Our current discount is $discount!";
```

You may also provide a block to the `package` declaration. Because this introduces a new block, it also provides a new lexical scope:

```
package Store::Toy {
    my $discount = 0.10;
}

package Store::Music {
    # $discount not visible
}

package Store::BoardGame;

# $discount still not visible
```

Variable Sigils

The sigil of the variable in a declaration determines the type of the variable: scalar, array, or hash. The sigil used when *accessing* a variable varies depending on what you do to the variable. For example, you declare an array as `@values`. Access the first element--a single value--of the array with

`$values[0]`. Access a list of values from the array with `@values[ @indices ]`. The sigil you use determines amount context in an lvalue situation:

```
# imposes lvalue context on some_function()
@values[ @indexes ] = some_function();
```

... or gets coerced in an rvalue situation:

```
# list evaluated to final element in scalar context
my $element = @values[ @indices ];
```

Anonymous Variables

Perl variables do not *require* names. Names exist to help you, the programmer, keep track of an `$apple`, `@barrels`, or `%cookie_recipes`. Variables created *without* literal names in your source code are *anonymous*. The only way to access anonymous variables is by reference (*references*).

Variables, Types, and Coercion

This relationship between variable types, sigils, and context is essential to your understanding of Perl.

A Perl variable represents both a value (a dollar cost, available pizza toppings, the names and numbers of guitar stores) and the container which stores that value. Perl's type system deals with *value types* and *container types*. While a variable's *container type*--scalar, array, or hash--cannot change, Perl is flexible about a variable's value type. You may store a string in a variable in one line, append to that variable a number on the next, and reassign a reference to a function (*function_references*) on the third, though this is a great way to confuse yourself.

Performing an operation on a variable which imposes a specific value type may cause coercion (*coercion*) of the variable's existing value type.

For example, the documented way to determine the number of entries in an array is to evaluate that array in scalar context (*context_philosophy*). Because a scalar variable can only ever contain a scalar, assigning an array (the rvalue) to a scalar (the lvalue) imposes scalar context on the operation, and an array evaluated in scalar context produces the number of elements in the array:

```
my $count = @items;
```

Values

New programmers spend a lot of time thinking about *what* their programs must do. Mature programmers spend their time designing a model for the data their programs must understand.

Variables allow you to manipulate data in the abstract. The values held in variables make programs concrete and useful. These values are your aunt's name and address, the distance between your office and a golf course on the moon, or the sum of the masses of all of the cookies you've eaten in the past year. Within your program, the rules regarding the format of that data are often strict.

Effective programs need effective (simple, fast, efficient, easy) ways to represent their data.

Strings

A *string* is a piece of textual or binary data with no particular formatting or contents. It could be your name, an image read from disk, or the source code of the program itself. A string has meaning in the program only when you give it meaning.

A literal string appears in your program surrounded by a pair of quoting characters. The most common *string delimiters* are single and double quotes:

```
my $name    = B<'Donner Odinson, Bringer of Despair'>;
my $address = B<"Room 539, Bilskirnir, Valhalla">;
```

Characters in a *single-quoted string* are exactly and only ever what they appear to be, with two exceptions. To include a single quote inside a single-quoted string, you must escape it with a leading backslash:

```
my $reminder = 'DonB<\'>t forget to escape '
             . 'the single quote!';
```

To include a backslash at the *end* of a string, escape it with another leading backslash. Otherwise Perl will think you're trying to escape the closing delimiter:

```
my $exception = 'This string ends with a '
              . 'backslash, not a quote: B<\\>';
```

Any other backslash will be part of the string as it appears, unless you have two adjacent backslashes, in which case Perl will believe that you intended to escape the second:

```
use Test::More;

is 'Modern B<\> Perl', 'Modern B<\\> Perl',
   'single quotes backslash escaping';

done_testing();
```

This example uses `Test::More` to prove the assertion that Perl considers these two lines equivalent. See *testing* for details on how that works.

A *double-quoted string* gives you more options, such as encoding otherwise invisible whitespace characters in the string:

```
my $tab       = "B<\t>";
my $newline   = "B<\n>";
my $carriage  = "B<\r>";
my $formfeed  = "B<\f>";
my $backspace = "B<\b>";
```

You may have inferred from this that you can represent the same logical string in multiple ways. You can include a tab within a string by typing the `\t` escape sequence or by hitting the Tab key on your keyboard. Both strings look and behave the same to Perl, even though the representation of the string may differ in the source code.

A string declaration may cross (and include) newlines, so these two declarations are equivalent:

```
my $escaped = "two\nlines";
my $literal = "two
lines";
is $escaped, $literal, 'equivalent \n and newline';
```

... but the escape sequences are easier for humans to read.

Perl strings have variable--not fixed--lengths. Perl will change their sizes for you as you modify and manipulate them. Use the *concatenation* operator `.` to combine multiple strings together:

```
    my $kitten = 'Choco' . ' ' . 'Spidermonkey';
```

... though concatenating three literal strings like this is ultimate the same to Perl as writing a single string.

When you *interpolate* the value of a scalar variable or the values of an array within a double-quoted string, the *current* contents of the variable become part of the string as if you'd concatenated them:

```
    my $factoid = "B<$name> lives at B<$address>!";

    # equivalent to
    my $factoid = $name . ' lives at ' . $address . '!';
```

Include a literal double-quote inside a double-quoted string by *escaping* it with a leading backslash:

```
    my $quote = "\"Ouch,\", he cried. \"That I<hurt>!\"";
```

Repeated backslashing sometimes becomes unwieldy. A *quoting operator* allows you to choose an alternate string delimiter. The q operator indicates single quoting (no interpolation), while the qq operator provides double quoting behavior (interpolation). The character immediately following the operator determines the characters used as delimiters. If the character is the opening character of a balanced pair--such as opening and closing braces--the closing character will be the final delimiter. Otherwise, the character itself will be both the starting and ending delimiter.

```
    my $quote     = B<qq{>"Ouch", he said. "That I<hurt>!"B<}>;
    my $reminder  =  B<q^>Don't escape the single quote!B<^>;
    my $complaint =  B<q{>It's too early to be awake.B<}>;
```

Use the *heredoc* syntax to assign multiple lines to a string:

```
    my $blurb =<<'END_BLURB';

    He looked up. "Change is the constant on which they all
    can agree.  We instead, born out of time, remain perfect
    and perfectly self-aware. We only suffer change as we
    pursue it. It is against our nature. We rebel against
    that change. Shall we consider them greater for it?"
    END_BLURB
```

This syntax has three parts. The double angle-brackets introduce the heredoc. The quotes determine whether the heredoc follows single- or double-quoted behavior; double-quoted behavior is the default. `END_BLURB` is an arbitrary identifier, chosen by the programmer, used as the ending delimiter.

Regardless of the indentation of the heredoc declaration itself, the ending delimiter must *start* at the beginning of the line:

```
    sub some_function {
        my $ingredients =<<'END_INGREDIENTS';
        Two eggs
        One cup flour
        Two ounces butter
        One-quarter teaspoon salt
        One cup milk
        One drop vanilla
        Season to taste
    B<END_INGREDIENTS>
    }
```

If the identifier *begins* with whitespace, that same whitespace must be present before the ending delimiter--that is, `<<' END_HEREDOC'>>` needs a leading space before `END_HEREDOC`. Yet if you indent the identifier, Perl will *not* remove equivalent whitespace from the start of each line of the heredoc. Keep that design wart in mind; it'll eventually surprise you.

Using a string in a non-string context will induce coercion (*coercion*).

Unicode and Strings

Unicode is a system used to represent the characters of the world's written languages. Most English text uses a character set of only 127 characters (which requires seven bits of storage and fits nicely into eight-bit bytes), but it's naïve to believe that you won't someday need an umlaut.

Perl strings can represent either of two separate but related data types:

Sequences of Unicode characters

Each character has a *codepoint*, a unique number which identifies it in the Unicode character set.

Sequences of octets

Binary data in a sequence of *octets*--8 bit numbers, each of which can represent a number between 0 and 255.

Why *octet* and not *byte*? An octet is unambiguously eight bits. A byte can be fewer or more bits, depending on esoteric hardware. Assuming that one character fits in one byte will cause you no end of Unicode grief. Separate the idea of memory storage from character representation. Forget that you ever heard of bytes.

Unicode strings and binary strings look superficially similar. Each has a `length()`. Each supports standard string operations such as concatenation, splicing, and regular expression processing (*chp.regex*). Any string which is not purely binary data is textual data, and thus should be a sequence of Unicode characters.

However, because of how your operating system represents data on disk or from users or over the network--as sequences of octets--Perl can't know if the data you read is an image file or a text document or anything else. By default, Perl treats all incoming data as sequences of octets. It's up to you to give that data meaning.

Character Encodings

A Unicode string is a sequence of octets which represents a sequence of characters. A *Unicode encoding* maps octet sequences to characters. Some encodings, such as UTF-8, can encode all of the characters in the Unicode character set. Other encodings represent only a subset of Unicode characters. For example, ASCII encodes plain English text (no accented characters allowed), while Latin-1 can represent text in most languages which use the Latin alphabet (umlauts, grave and circumflex accents, et cetera).

Perl 5.16 supports the Unicode 6.1 standard, 5.18 the 6.2 standard, 5.20 the 6.3 standard, and 5.22 the 7.0 standard. See http://unicode.org/versions/.

To avoid most Unicode problems, always decode to and from the appropriate encoding at the inputs and outputs of your program. Read that sentence again. Memorize it. You'll be glad of it later.

Unicode in Your Filehandles

When you tell Perl that a specific filehandle (*files*) should interpret data via specific Unicode encoding, Perl will use an *IO layer* to convert between octets and characters. The *mode* operand of the `open` builtin allows you to request an IO layer by name. For example, the `:utf8` layer decodes UTF-8 data:

```
open my $fh, '<:utf8', $textfile;

my $unicode_string = <$fh>;
```

Use `binmode` to apply an IO layer to an existing filehandle:

```
binmode $fh, ':utf8';
my $unicode_string = <$fh>;

binmode STDOUT, ':utf8';
say $unicode_string;
```

If you want to write Unicode to files, you must specify the desired encoding. Otherwise, Perl will warn you when you print Unicode characters that don't look like octets; this is what `Wide character in %s` means.

Use the `utf8::all` module to add the UTF-8 IO layer to all filehandles throughout your program. The module also enables all sorts of other Unicode features. It's very handy, but it's a blunt instrument and no substitute for understanding what your program needs to do.

Unicode in Your Data

The core module `Encode`'s `decode()` function converts a sequence of octets to Perl's internal Unicode representation. The corresponding `encode()` function converts from Perl's internal encoding to the desired encoding:

```
my $from_utf8 = decode('utf8', $data);
my $to_latin1 = encode('iso-8859-1', $string);
```

To handle Unicode properly, you must always *decode* incoming data via a known encoding and *encode* outgoing data to a known encoding. Again, you must know what kind of data you expect to consume and to produce. Being specific will help you avoid all kinds of trouble.

Unicode in Your Programs

The easiest way to use Unicode characters in your source code us with the `utf8` pragma (*pragmas*), which tells the Perl parser to decode the rest of the file as UTF-8 characters. This allows you to use

Unicode characters in strings and identifiers:

```
use utf8;

sub E<pound>_to_E<yen> { ... }

my $yen = E<pound>_to_E<yen>('1000E<pound>');
```

To *write* this code, your text editor must understand UTF-8 and you must save the file with the appropriate encoding. Again, any two programs which communicate with Unicode data must agree on the encoding of that data.

Within double-quoted strings, you may also use a Unicode escape sequence to represent character encodings. The syntax `\x{}` represents a single character; place the hex form of the character's Unicode numberhttp://unicode.org/charts/ within the curly brackets:

```
my $escaped_thorn = "\x{00FE}";
```

Some Unicode characters have names, which make them easier for other programmers to read. Use the `charnames` pragma to enable named characters via the `\N{}` escape syntax:

```
use charnames ':full';
use Test::More tests => 1;

my $escaped_thorn = "\x{00FE}";
my $named_thorn   = "\N{LATIN SMALL LETTER THORN}";

is $escaped_thorn, $named_thorn,
    'Thorn equivalence check';
```

You may use the `\x{}` and `\N{}` forms within regular expressions as well as anywhere else you may legitimately use a string or a character.

Implicit Conversion

Most Unicode problems in Perl arise from the fact that a string could be *either* a sequence of octets *or* a sequence of characters. Perl allows you to combine these types through the use of implicit conversions. When these conversions are wrong, they're rarely *obviously* wrong but they're also often *spectacularly* wrong in ways that are difficult to debug.

When Perl concatenates a sequence of octets with a sequence of Unicode characters, it implicitly decodes the octet sequence using the Latin-1 encoding. The resulting string will contain Unicode characters. When you print Unicode characters, Perl will encode the string using UTF-8, because Latin-1 cannot represent the entire set of Unicode characters--because Latin-1 is a subset of UTF-8.

The asymmetry between encodings and octets can lead to Unicode strings encoded as UTF-8 for output and decoded as Latin-1 from input. Worse yet, when the text contains only English characters with no accents, the bug stays hidden, because both encodings use the same representation for every character.

You don't have to understand all of this right now; just know that this behavior happens and that it's not what you want.

```
my $hello    = "Hello, ";
my $greeting = $hello . $name;
```

If `$name` contains *Alice*, you will never notice any problem: because the Latin-1 representation is the same as the UTF-8 representation. If `$name` contains *José*, `$name` can contain several possible values:

- `$name` contains four Unicode characters.
- `$name` contains four Latin-1 octets representing four Unicode characters.
- `$name` contains *five* UTF-8 octets representing four Unicode characters.

The string literal has several possible scenarios:

* It is an ASCII string literal and contains octets: `my $hello = "Hello, ";`

* It is a Latin-1 string literal with no explicit encoding and contains octets: `my $hello = "¡Hola, ";`

* It is a non-ASCII string literal (the `utf8` or `encoding` pragma is in effect) and contains Unicode characters: `my $hello = "Kuirabá, ";`

If both `$hello` and `$name` are Unicode strings, the concatenation will produce another Unicode string.

If both strings are octet sequences, Perl will concatenate them into a new octet sequence. If both values are octets of the same encoding--both Latin-1, for example, the concatenation will work correctly. If the octets do not share an encoding--for example, a concatenation appending UTF-8 data to Latin-1 data--then the resulting sequence of octets makes sense in *neither* encoding. This could happen if the user entered a name as UTF-8 data and the greeting were a Latin-1 string literal, but the program decoded neither.

If only one of the values is a Unicode string, Perl will decode the other as Latin-1 data. If this is not the correct encoding, the resulting Unicode characters will be wrong. For example, if the user input were UTF-8 data and the string literal were a Unicode string, the name would be incorrectly decoded into five Unicode characters to form *JosÃ©* (*sic*) instead of *José* because the UTF-8 data means something else when decoded as Latin-1 data.

Again, you don't have to follow all of the details here if you remember this: always decode on input and encode on output.

See `perldoc perluniintro` for a far more detailed explanation of Unicode, encodings, and how to manage incoming and outgoing data in a Unicode world. For *far* more detail about managing Unicode effectively throughout your programs, see Tom Christiansen's answer to "Why does Modern Perl avoid UTF-8 by default?" http://stackoverflow.com/questions/6162484/why-does-modern-perl-avoid-utf-8-by-default/6163129#6163129 and his "Perl Unicode Cookbook" series http://www.perl.com/pub/2012/04/perlunicook-standard-preamble.html.

If you work with Unicode in Perl, use at least Perl 5.18 (and ideally the latest version). See also the `feature` pragma for information on the `unicode_strings` feature.

Numbers

Perl supports numbers as both integers and floating-point values. You may represent them with scientific notation as well as in binary, octal, and hexadecimal forms:

```
    my $integer   = 42;
    my $float     = 0.007;
    my $sci_float = 1.02e14;
    my $binary    = B<0b>101010;
    my $octal     = B<0>52;
    my $hex       = B<0x>20;

    # only in Perl 5.22
    my $hex_float = B<0x>1.0p-3;
```

The numeric prefixes `0b`, `0`, and `0x` specify binary, octal, and hex notation respectively. Be aware that a leading zero on an integer *always* indicates octal mode.

Even though you can write floating-point values explicitly with perfect accuracy, Perl--like most programming languages--represents them internally in a binary format. This representation is sometimes imprecise in specific ways; consult `perldoc perlnumber` for more details. Perl 5.22 allows you to use a hexadecimal representation of floating point values, so as to keep maximum precision. See "Scalar value constructors" in `perldoc perldata` for more information.

You may *not* use commas to separate thousands in numeric literals, as the parser will interpret them as the comma operator. Instead, use underscores. These three examples are equivalent, though the second might be the most readable:

```
    my $billion = 1000000000;
    my $billion = 1_000_000_000;
    my $billion = 10_0_00_00_0_0_0;
```

Because of coercion (*coercion*), Perl programmers rarely have to worry about converting incoming data to numbers. Perl will treat anything which looks like a number *as* a number when evaluated in a numeric context. In the rare circumstances where *you* need to know if something looks like a number without evaluating it in a numeric context, use the `looks_like_number` function from the core module `Scalar::Util`. This function returns a true value if Perl will consider the given argument numeric.

The `Regexp::Common` module from the CPAN provides several well-tested regular expressions to identify more specific *types* of numeric values such as whole numbers, integers, and floating-point values.

What's the maximum size of a value you can represent in Perl? It depends; you're probably using a 64-bit build, so the largest integer is `(2**31) - 1` and the smallest is `-(2**31)`--though see `perldoc perlnumber` for more thorough details. Use `Math::BigInt` and `Math::BigFloat` to handle with larger or smaller or more precise numbers.

Undef

Perl's `undef` value represents an unassigned, undefined, and unknown value. Declared but undefined scalar variables contain `undef`:

```
    my $name = undef;   # unnecessary assignment
    my $rank;           # also contains undef
```

`undef` evaluates to false in boolean a context. Evaluating `undef` in a string context--such as interpolating it into a string:

```
my $undefined;
my $defined = $undefined . '... and so forth';
```

... produces an `uninitialized value` warning:

```
Use of uninitialized value $undefined in
concatenation (.) or string...
```

The `defined` builtin returns a true value if its operand evaluates to a defined value (anything other than `undef`):

```
my $status = 'suffering from a cold';

say B<defined> $status;  # 1, which is a true value
say B<defined> undef;    # empty string; a false value
```

The Empty List

When used on the right-hand side of an assignment, the `()` construct represents an empty list. In scalar context, this evaluates to `undef`. In list context, it is an empty list. When used on the left-hand side of an assignment, the `()` construct imposes list context. Hence this idiom (*idioms*) to count the number of elements returned from an expression in list context without using a temporary variable:

```
my $count = B<()> = get_clown_hats();
```

Because of the right associativity (*associativity*) of the assignment operator, Perl first evaluates the second assignment by calling `get_clown_hats()` in list context. This produces a list.

Assignment to the empty list throws away all of the values of the list, but that assignment takes place in scalar context, which evaluates to the number of items on the right hand side of the assignment. As a result, `$count` contains the number of elements in the list returned from `get_clown_hats()`.

This idiom often confuses new programmers, but with practice, you'll understand how Perl's fundamental design features fit together.

Lists

A list is a comma-separated group of one or more expressions. Lists may occur verbatim in source code as values:

```
my @first_fibs = B<(1, 1, 2, 3, 5, 8, 13, 21);>
```

... as targets of assignments:

```
my B<($package, $filename, $line)> = caller();
```

... or as lists of expressions:

```
say B<< name(), ' => ', age() >>;
```

Parentheses do not *create* lists. The comma operator creates lists. The parentheses in these examples merely group expressions to change their *precedence* (*precedence*).

As an example of lists without parens, use the range operator to create lists of literals in a compact form:

```
my @chars = 'a' .. 'z';
my @count = 13 .. 27;
```

Use the qw() operator to split a literal string on whitespace to produce a list of strings. As this is a quoting operator, you may choose any delimiters you like:

```
my @stooges = qw! Larry Curly Moe Shemp Joey Kenny !;
```

Perl will emit a warning if a qw() contains a comma or the comment character (#), because not only are such characters rare in a qw(), their presence is often a mistake.

Lists can (and often do) occur as the results of expressions, but these lists do not appear literally in source code.

Lists and arrays are not interchangeable in Perl. Lists are values. Arrays are containers. You may store a list in an array and you may coerce an array to a list, but they are separate entities. For example, indexing into a list always occurs in list context. Indexing into an array can occur in scalar context (for a single element) or list context (for a slice):

```
# don't worry about the details right now
sub context
{
    my $context = wantarray();

    say defined $context
         ? $context
             ? 'list'
             : 'scalar'
         : 'void';
    return 0;
}

my @list_slice  = (1, 2, 3)[context()];
my @array_slice = @list_slice[context()];
my $array_index = $array_slice[context()];

say context(); # list context
context();     # void context
```

Control Flow

Perl's basic *control flow* is straightforward. Program execution starts at the beginning (the first line of the file executed) and continues to the end:

```
say 'At start';
say 'In middle';
say 'At end';
```

Perl's *control flow directives* change the order of what happens next in the program.

Branching Directives

The `if` directive performs the associated action only when its conditional expression evaluates to a *true* value:

```
say 'Hello, Bob!' if $name eq 'Bob';
```

This postfix form is useful for simple expressions. Its block form groups multiple expressions into a unit which evaluates to a single boolean value:

```
if ($name eq 'Bob') {
    say 'Hello, Bob!';
    found_bob();
}
```

The conditional expression may consist of multiple subexpressions which will be coerced to a boolean value:

```
if ($name eq 'Bob' && not greeted_bob()) {
    say 'Hello, Bob!';
    found_bob();
}
```

The block form requires parentheses around its condition, but the postfix form does not. In the postfix form, adding parentheses can clarify the intent of the code at the expense of visual cleanliness:

```
greet_bob() if ($name eq 'Bob' && not greeted_bob());
```

The `unless` directive is the negated form of `if`. Perl will perform the action when the conditional expression evaluates to a *false* value:

```
say "You're not Bob!" unless $name eq 'Bob';
```

Like `if`, `unless` also has a block form, though many programmers avoid it due to its potential for confusion:

```
unless (is_leap_year() and is_full_moon()) {
    frolic();
    gambol();
}
```

`unless` works very well for postfix conditionals, especially parameter validation in functions (*postfix_parameter_validation*):

```
sub frolic {
    # do nothing without parameters
    return unless @_;

    for my $chant (@_) { ... }
}
```

The block forms of `if` and `unless` both support the `else` directive, which provides a block to execute when the conditional expression does not evaluate to the appropriate value:

```
    if ($name eq 'Bob') {
        say 'Hi, Bob!';
        greet_user();
    }
    else {
        say "I don't know you.";
        shun_user();
    }
```

`else` blocks allow you to rewrite `if` and `unless` conditionals in terms of each other:

```
    unless ($name eq 'Bob') {
        say "I don't know you.";
        shun_user();
    }
    else {
        say 'Hi, Bob!';
        greet_user();
    }
```

However, the implied double negative of using `unless` with an `else` block can be confusing. This example may be the only place you ever see it.

Just as Perl provides both `if` and `unless` to allow you to phrase your conditionals in the most readable way, Perl has both positive and negative conditional operators:

```
    if ($name B<ne> 'Bob') {
        say "I don't know you.";
        shun_user();
    }
    else {
        say 'Hi, Bob!';
        greet_user();
    }
```

... though the double negative implied by the presence of the `else` block may be difficult to read.

Use one or more `elsif` directives to check multiple and mutually exclusive conditions:

```
    if ($name eq 'Robert') {
        say 'Hi, Bob!';
        greet_user();
    }
    elsif ($name eq 'James') {
        say 'Hi, Jim!';
        greet_user();
    }
    elsif ($name eq 'Armando') {
        say 'Hi, Mando!';
        greet_user();
    }
    else {
        say "You're not my uncle.";
        shun_user();
    }
```

An `unless` chain may also use an `elsif` block, but good luck deciphering that.

Perl supports neither `elseunless` nor `else if`. Larry prefers `elsif` for aesthetic reasons, as well the prior art of the Ada programming language:

```
if ($name eq 'Rick') {
    say 'Hi, cousin!';
}

# warning; syntax error
else if ($name eq 'Kristen') {
    say 'Hi, cousin-in-law!';
}
```

The Ternary Conditional Operator

The *ternary conditional* operator evaluates a conditional expression and evaluates to one of two alternatives:

```
my $time_suffix = after_noon($time)
                ? 'afternoon'
                : 'morning';

# equivalent to
my $time_suffix;

if (after_noon(time)) {
    $time_suffix = 'afternoon';
}
else {
    $time_suffix = 'morning';
}
```

The conditional expression precedes the question mark character (`?`). The colon character (`:`) separates the alternatives. The alternatives are expressions of arbitrary complexity--including other ternary conditional expressions, though consider clarity over concision.

An interesting, though obscure, idiom uses the ternary conditional to select between alternative *variables*, not only values:

```
push @{ rand() > 0.5 ? \@red_team : \@blue_team },
    Player->new;
```

Short Circuiting

Perl exhibits *short-circuiting* behavior when it encounters complex conditional expressions. When Perl can determine that a complex expression would succeed or fail as a whole without evaluating every subexpression, it will not evaluate subsequent subexpressions. This is most obvious with an example:

```
say 'Both true!' if ok 1, 'subexpression one'
                 && ok 1, 'subexpression two';

done_testing();
```

The return value of `ok()` (*testing*) is the boolean value produced by the first argument, so the example prints:

```
ok 1 - subexpression one
ok 2 - subexpression two
Both true!
```

When the first subexpression--the first call to `ok`--evaluates to a true value, Perl must evaluate the second subexpression. If the first subexpression had evaluated to a false value, there would be no need to check subsequent subexpressions, as the entire expression could not succeed:

```
say 'Both true!' if ok 0, 'subexpression one'
                 && ok 1, 'subexpression two';
```

This example prints:

```
not ok 1 - subexpression one
```

Even though the second subexpression would obviously succeed, Perl never evaluates it. The same short-circuiting behavior is evident for logical-or operations:

```
say 'Either true!' if ok 1, 'subexpression one'
                   || ok 1, 'subexpression two';
```

This example prints:

```
ok 1 - subexpression one
Either true!
```

Given the success of the first subexpression, Perl can avoid evaluating the second subexpression. If the first subexpression were false, the result of evaluating the second subexpression would dictate the result of evaluating the entire expression.

Besides allowing you to avoid potentially expensive computations, short circuiting can help you to avoid errors and warnings, as in the case where using an undefined value might raise a warning:

```
my $bbq;
if (defined $bbq and $bbq eq 'brisket') { ... }
```

Context for Conditional Directives

The conditional directives--`if`, `unless`, and the ternary conditional operator--all evaluate an expression in boolean context (*context_philosophy*). As comparison operators such as `eq`, `==`, `ne`, and `!=` all produce boolean results when evaluated, Perl coerces the results of other expressions--including variables and values--into boolean forms.

Perl has neither a single true value nor a single false value. Any number which evaluates to 0 is false. This includes `0`, `0.0`, `0e0`, `0x0`, and so on. The empty string (`''`) and `'0'` evaluate to a false value, but the strings `'0.0'`, `'0e0'`, and so on do not. The idiom `'0 but true'` evaluates to 0 in numeric context--but true in boolean context due to its string contents.

Both the empty list and `undef` evaluate to a false value. Empty arrays and hashes return the number 0 in scalar context, so they evaluate to a false value in boolean context. An array which contains a single element--even `undef`--evaluates to true in boolean context. A hash which contains any elements--even a key and a value of `undef`--evaluates to a true value in boolean context.

The `Want` module from the CPAN allows you to detect boolean context within your own functions. The core `overloading` pragma (*overloading*) allows you to specify what your own data types produce when evaluated in various contexts.

Looping Directives

Perl provides several directives for looping and iteration. The *foreach*-style loop evaluates an expression which produces a list and executes a statement or block until it has exhausted that list:

```
# square the first ten positive integers
foreach (1 .. 10) {
    say "$_ * $_ = ", $_ * $_;
}
```

This example uses the range operator to produce a list of integers from one to ten inclusive. The `foreach` directive loops over them, setting the topic variable $_ (*default_scalar_variable*) to each in turn. Perl executes the block for each integer and, as a result, prints the squares of the integers.

Many Perl programmers refer to iteration as `foreach` loops, but Perl treats the names `foreach` and `for` interchangeably. The parenthesized expression determines the type and behavior of the loop; the keyword does not.

Like `if` and `unless`, this loop has a postfix form:

```
say "$_ * $_ = ", $_ * $_ for 1 .. 10;
```

A `for` loop may use a named variable instead of the topic:

```
for my $i (1 .. 10) {
    say "$i * $i = ", $i * $i;
}
```

When a `for` loop uses an iterator variable, the variable is scoped to the block *within* the loop. Perl will set this lexical to the value of each item in the iteration. Perl will not modify the topic variable ($_). If you have declared a lexical `$i` in an outer scope, its value will persist outside the loop:

```
my $i = 'cow';

for my $i (1 .. 10) {
    say "$i * $i = ", $i * $i;
}

is $i, 'cow', 'Value preserved in outer scope';
```

This localization occurs even if you do not redeclare the iteration variable as a lexical, but keep the habit of declaring iteration values as lexicals:

```
my $i = 'horse';

for $i (1 .. 10) {
    say "$i * $i = ", $i * $i;
}

is $i, 'horse', 'Value preserved in outer scope';
```

Iteration and Aliasing

The `for` loop *aliases* the iterator variable to the values in the iteration such that any modifications to the value of the iterator modifies the value in place:

```
my @nums = 1 .. 10;

$_ **= 2 for @nums;

is $nums[0], 1, '1 * 1 is 1';
is $nums[1], 4, '2 * 2 is 4';

...

is $nums[9], 100, '10 * 10 is 100';
```

This aliasing also works with the block style `for` loop:

```
for my $num (@nums) {
    $num **= 2;
}
```

... as well as iteration with the topic variable:

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

You cannot use aliasing to modify *constant* values, however. Perl will produce an exception about modification of read-only values.

```
$_++ and say for qw( Huex Dewex Louid );
```

You may occasionally see the use of `for` with a single scalar variable:

```
for ($user_input) {
    s/\A\s+//;      # trim leading whitespace
    s/\s+\z//;      # trim trailing whitespace

    $_ = quotemeta; # escape non-word characters
}
```

This idiom (*idioms*) uses the iteration operator for its side effect of aliasing $_, though it's clearer to operate on the named variable itself.

Iteration and Scoping

The topic variable's iterator scoping has a subtle gotcha. Consider a function `topic_mangler()` which modifies $_ on purpose. If code iterating over a list called `topic_mangler()` without protecting $_, you'd have to spend some time debugging the effects:

```
for (@values) {
    topic_mangler();
}
```

```
sub topic_mangler {
    s/foo/bar/;
}
```

The substitution in `topic_mangler()` will modify elements of `@values` in place. If you *must* use `$_` rather than a named variable, use the topic aliasing behavior of `for`:

```
sub topic_mangler {
    # was $_ = shift;
    B<for (shift)>
    {
        s/foo/bar/;
        s/baz/quux/;
        return $_;
    }
}
```

Alternately, use a named iteration variable in the `for` loop. That's almost always the right advice.

The C-Style For Loop

The C-style *for loop* requires you to manage the conditions of iteration:

```
for (my $i = 0; $i <= 10; $i += 2) {
    say "$i * $i = ", $i * $i;
}
```

You must explicitly assign to an iteration variable in the looping construct, as this loop performs neither aliasing nor assignment to the topic variable. While any variable declared in the loop construct is scoped to the lexical block of the loop, Perl will not limit the lexical scope of a variable declared outside of the loop construct:

```
my $i = 'pig';

for ($i = 0; $i <= 10; $i += 2) {
    say "$i * $i = ", $i * $i;
}

isnt $i, 'pig', '$i overwritten with a number';
```

The looping construct may have three subexpressions. The first subexpression--the initialization section--executes only once, before the loop body executes. Perl evaluates the second subexpression--the conditional comparison--before each iteration of the loop body. When this evaluates to a true value, iteration proceeds. When it evaluates to a false value, iteration stops. The final subexpression executes after each iteration of the loop body.

```
for (
    # loop initialization subexpression
    say 'Initializing', my $i = 0;

    # conditional comparison subexpression
    say "Iteration: $i" and $i < 10;

    # iteration ending subexpression
    say 'Incrementing ' . $i++
) {
    say "$i * $i = ", $i * $i;
}
```

Note the lack of a semicolon after the final subexpression as well as the use of the comma operator and low-precedence `and`; this syntax is surprisingly finicky. When possible, prefer the `foreach`-style loop to the `for` loop.

All three subexpressions are optional. One infinite `for` loop is:

```
    for (;;) { ... }
```

While and Until

A *while* loop continues until the loop conditional expression evaluates to a false value. An idiomatic infinite loop is:

```
    while (1) { ... }
```

Unlike the iteration `foreach`-style loop, the `while` loop's condition has no side effects. If `@values` has one or more elements, this code is also an infinite loop, because every iteration will evaluate `@values` in scalar context to a non-zero value and iteration will continue:

```
    while (@values) {
        say $values[0];
    }
```

To prevent such an infinite `while` loop, use a *destructive update* of the `@values` array by modifying the array within each iteration:

```
    while (@values) {
        my $value = shift @values;
        say $value;
    }
```

Modifying `@values` inside of the `while` condition check also works, but it has some subtleties related to the truthiness of each value.

```
    while (my $value = shift @values) {
        say $value;
    }
```

This loop will exit as soon as *the assignment expression* used as the conditional expression evaluates to a false value. If that's what you intend, add a comment to the code.

The *until* loop reverses the sense of the test of the `while` loop. Iteration continues while the loop conditional expression evaluates to a false value:

```
    until ($finished_running) {
        ...
    }
```

The canonical use of the `while` loop is to iterate over input from a filehandle:

```
    while (<$fh>) {
        # remove newlines
        chomp;
        ...
    }
```

Perl interprets this `while` loop as if you had written:

```
    while (B<defined($_> = <$fh>B<)>) {
        # remove newlines
        chomp;
        ...
    }
```

Without the implicit `defined`, any line read from the filehandle which evaluated to a false value in a scalar context--a blank line or a line which contained only the character `0`--would end the loop. The `readline` (<>) operator returns an undefined value only when it has reached the end of the file.

Both `while` and `until` have postfix forms, such as the infinite loop `1 while 1;`. Any single expression is suitable for a postfix `while` or `until`, including the classic "Hello, world!" example from 8-bit computers of the early 1980s:

```
    print "Hello, world!  " while 1;
```

Infinite loops are more useful than they seem, especially for event loops in GUI programs, program interpreters, or network servers:

```
    $server->dispatch_results until $should_shutdown;
```

Use a do block to group several expressions into a single unit:

```
    do {
        say 'What is your name?';
        my $name = <>;
        chomp $name;
        say "Hello, $name!" if $name;
    } until (eof);
```

A `do` block parses as a single expression which may contain several expressions. Unlike the `while` loop's block form, the `do` block with a postfix `while` or `until` will execute its body *at least* once. This construct is less common than the other loop forms, but very powerful.

Loops within Loops

You may nest loops within other loops:

```
    for my $suit (@suits) {
        for my $values (@card_values) { ... }
    }
```

Note the value of declaring iteration variables! The potential for confusion with the topic variable and its scope is too great otherwise.

Novices commonly exhaust filehandles accidentally while nesting `foreach` and `while` loops:

```
    use autodie 'open';
    open my $fh, '<', $some_file;
```

```
for my $prefix (@prefixes) {

    # DO NOT USE; buggy code
    while (<$fh>) {
        say $prefix, $_;
    }
}
```

Opening the filehandle outside of the `for` loop leaves the file position unchanged between each iteration of the `for` loop. On its second iteration, the `while` loop will have nothing to read (the `readline` will return a false value). You can solve this problem in many ways; re-open the file inside the `for` loop (wasteful but simple), slurp the entire file into memory (works best with small files), or `seek` the filehandle back to the beginning of the file for each iteration:

```
for my $prefix (@prefixes) {
    while (<$fh>) {
        say $prefix, $_;
    }

    B<seek $fh, 0, 0;>
}
```

Loop Control

Sometimes you must break out of a loop before you have exhausted the iteration conditions. Perl's standard control mechanisms--exceptions and `return`--work, but you may also use *loop control* statements.

The *next* statement restarts the loop at its next iteration. Use it when you've done everything you need to in the current iteration. To loop over lines in a file and skip everything that starts with the comment character #:

```
while (<$fh>) {
    B<next> if /\A#/;
    ...
}
```

Compare the use of `next` with the alternative: wrapping the rest of the body of the block in an `if`. Now consider what happens if you have multiple conditions which could cause you to skip a line. Loop control modifiers with postfix conditionals can make your code much more readable.

The *last* statement ends the loop immediately. To finish processing a file once you've seen the ending token, write:

```
while (<$fh>) {
    next if /\A#/;
    B<last> if /\A__END__/
    ...
}
```

The *redo* statement restarts the current iteration without evaluating the conditional again. This can be useful in those few cases where you want to modify the line you've read in place, then start processing over from the beginning without clobbering it with another line. To implement a silly file parser that joins lines which end with a backslash:

```
while (my $line = <$fh>) {
    chomp $line;

    # match backslash at the end of a line
    if ($line =~ s{\\$}{})
    {
        $line .= <$fh>;
        B<redo;>
    }

    ...
}
```

Nested loops can be confusing, especially with loop control statements. If you cannot extract inner loops into named functions, use *loop labels* to clarify your intent:

```
B<LINE:>
while (<$fh>) {
    chomp;

    B<PREFIX:>
    for my $prefix (@prefixes) {
        next LINE unless $prefix;
        say "$prefix: $_";
        # next PREFIX is implicit here
    }
}
```

Continue

The `continue` construct behaves like the third subexpression of a `for` loop; Perl executes any continue block before subsequent iterations of a loop, whether due to normal loop repetition or premature re-iteration from `next`. You may use it with a `while`, `until`, `when`, or `for` loop. Examples of `continue` are rare, but it's useful any time you want to guarantee that something occurs with every iteration of the loop, regardless of how that iteration ends:

```
while ($i < 10 ) {
    next unless $i % 2;
    say $i;
}
B<continue> {
    say 'Continuing...';
    $i++;
}
```

Be aware that a `continue` block does *not* execute when control flow leaves a loop due to `last` or `redo`.

Switch Statements

Perl 5.10 introduced a new construct named `given` as a Perlish `switch` statement. It didn't quite work out; `given` is still experimental, if less buggy in newer releases. Avoid it unless you know exactly what you're doing.

If you need a switch statement, use `for` to alias the topic variable (`$_`) and `when` to match it against simple expressions with smart match (*smart_match*) semantics. To write the Rock, Paper, Scissors game:

```
    my @options  = ( \&rock, \&paper, \&scissors );
    my $confused = "I don't understand your move.";

    do {
        say "Rock, Paper, Scissors!  Pick one: ";
        chomp( my $user = <STDIN> );
        my $computer_match = $options[ rand @options ];
        $computer_match->( lc( $user ) );
    } until (eof);

    sub rock {
        print "I chose rock.  ";

        for (shift) {
            when (/paper/)    { say 'You win!' };
            when (/rock/)     { say 'We tie!'  };
            when (/scissors/) { say 'I win!'   };
            default           { say $confused  };
        }
    }

    sub paper {
        print "I chose paper.  ";

        for (shift) {
            when (/paper/)    { say 'We tie!'  };
            when (/rock/)     { say 'I win!'   };
            when (/scissors/) { say 'You win!' };
            default           { say $confused  };
        }
    }

    sub scissors {
        print "I chose scissors.  ";

        for (shift) {
            when (/paper/)    { say 'I win!'   };
            when (/rock/)     { say 'You win!' };
            when (/scissors/) { say 'We tie!'  };
            default           { say $confused  };
        }
    }
```

Perl executes the `default` rule when none of the other conditions match. Adding Spock and Lizard is left as an exercise for the reader.

Tailcalls

A *tailcall* occurs when the last expression within a function is a call to another function. The outer function's return value becomes the inner function's return value:

```
sub log_and_greet_person {
    my $name = shift;
    log( "Greeting $name" );

    return greet_person( $name );
}
```

Returning from `greet_person()` directly to the caller of `log_and_greet_person()` is more efficient than returning *to* `log_and_greet_person()` and then *from* `log_and_greet_person()`. Returning directly *from* `greet_person()` to the caller of `log_and_greet_person()` is a *tailcall optimization*.

Heavily recursive code (*recursion*)--especially mutually recursive code--can consume a lot of memory. Tailcalls reduce the memory needed for internal bookkeeping of control flow and can make expensive algorithms cheaper. Unfortunately, Perl does not automatically perform this optimization, so you have to do it yourself when it's necessary.

The builtin `goto` operator has a form which calls a function as if the current function were never called, essentially erasing the bookkeeping for the new function call. The ugly syntax confuses people who've heard "Never use `goto`", but it works:

```
sub log_and_greet_person {
    B<my ($name) = @_;>
    log( "Greeting $name" );

    B<goto &greet_person>;
}
```

This example has two important characteristics. First, `goto &function_name` or `goto &$function_reference` requires the use of the function sigil (&) so that the parser knows to perform a tailcall instead of jumping to a label. Second, this form of function call passes the contents of `@_` implicitly to the called function. You may modify `@_` to change the passed arguments if you desire.

This technique is most useful when you want to hijack control flow to get out of the way of other functions inspecting `caller` (such as when you're implementing special logging or some sort of debugging feature), or when using an algorithm which requires a lot of recursion. Remember it if you need it, but feel free not to use it.

Scalars

Perl's fundamental data type is the *scalar*: a single, discrete value. That value may be a string, an integer, a floating point value, a filehandle, or a reference--but it is always a single value. Scalars may be lexical, package, or global (*globals*) variables. You may only declare lexical or package variables. The names of scalar variables must conform to standard variable naming guidelines (*names*). Scalar variables always use the leading dollar-sign ($) sigil (*sigils*).

Scalar values and scalar context have a deep connection; assigning to a scalar imposes scalar context. Using the scalar sigil with an aggregate variable accesses a single element of the hash or array in scalar context.

Scalars and Types

A scalar variable can contain any type of scalar value without special conversions, coercions, or casts. The type of value stored in a scalar variable, once assigned, can change arbitrarily:

```
my $value;
$value = 123.456;
$value = 77;
$value = "I am Chuck's big toe.";
$value = Store::IceCream->new;
```

Even though this code is *legal*, changing the type of data stored in a scalar is confusing.

This flexibility of type often leads to value coercion (*coercion*). For example, you may treat the contents of a scalar as a string, even if you didn't explicitly assign it a string:

```
my $zip_code       = 97123;
my $city_state_zip = 'Hillsboro, Oregon' . ' ' . $zip_code;
```

You may also use mathematical operations on strings:

```
my $call_sign = 'KBMIU';

# update sign in place and return new value
my $next_sign = ++$call_sign;

# return old value, I<then> update sign
my $curr_sign = $call_sign++;

# but I<does not work> as:
my $new_sign  = $call_sign + 1;
```

This magical string increment behavior has no corresponding magical decrement behavior. You can't restore the previous string value by writing `$call_sign--`.

This string increment operation turns a into b and z into aa, respecting character set and case. While ZZ9 becomes AAA0, ZZ09 becomes ZZ10--numbers wrap around while there are more significant places to increment, as on a vehicle odometer.

Evaluating a reference (*references*) in string context produces a string. Evaluating a reference in numeric context produces a number. Neither operation modifies the reference in place, but you cannot recreate the reference from either result:

```
my $authors     = [qw( Pratchett Vinge Conway )];
my $stringy_ref = '' . $authors;
my $numeric_ref =  0 + $authors;
```

`$authors` is still useful as a reference, but `$stringy_ref` is a string with no connection to the reference and `$numeric_ref` is a number with no connection to the reference.

To allow coercion without data loss, Perl scalars can contain both numeric and string components. The internal data structure which represents a scalar in Perl has a numeric slot and a string slot.

Accessing a string in a numeric context produces a scalar with both string and numeric values.

Scalars do not contain a separate slot for boolean values. In boolean context, the empty strings (`''`) and `'0'` evaluate to false values. All other strings evaluate to true values. In boolean context, numbers which evaluate to zero (`0`, `0.0`, and `0e0`) evaluate to false values. All other numbers evaluate to true values.

Be careful that the *strings* `'0.0'` and `'0e0'` evaluate to true values. This is one place where Perl makes a distinction between what *looks like* a number and what really is a number.

`undef` is always a false value.

Arrays

Perl's *array* data type is a language-supported aggregate which can store zero or more scalars. You can access individual members of the array by integer indexes, and you can add or remove elements at will. Arrays grow or shrink as you manipulate them.

The `@` sigil denotes an array. To declare an array:

```
my @items;
```

Array Elements

Use the scalar sigil to *access* an individual element of an array. `$cats[0]` refers unambiguously to the `@cats` array, because postfix (*fixity*) square brackets (`[]`) always mean indexed access to an array.

The first element of an array is at the zeroth index:

```
# @cats contains a list of Cat objects
my $first_cat = $cats[0];
```

The last index of an array depends on the number of elements in the array. An array in scalar context (due to scalar assignment, string concatenation, addition, or boolean context) evaluates to the number of elements in the array:

```
# scalar assignment
my $num_cats = @cats;

# string concatenation
say 'I have ' . @cats . ' cats!';

# addition
my $num_animals = @cats + @dogs + @fish;

# boolean context
say 'Yep, a cat owner!' if @cats;
```

To get the *index* of the final element of an array, subtract one from the number of elements of the array (because array indexes start at 0) or use the unwieldy `$#cats` syntax:

```
my $first_index = 0;
my $last_index  = @cats - 1;
# or
# my $last_index = $#cats;

say   "My first cat has an index of $first_index, "
    . "and my last cat has an index of $last_index."
```

Most of the time you care more about the relative position of an array element. Use a negative array index to refer to elements from the end. The last element of an array is available at the index `-1`. The second to last element of the array is available at index `-2`, and so on:

```
my $last_cat           = $cats[-1];
my $second_to_last_cat = $cats[-2];
```

`$#` has another use: resize an array in place by *assigning* to `$#array`. Remember that Perl arrays are mutable. They expand or contract as necessary. When you shrink an array, Perl will discard values which do not fit in the resized array. When you expand an array, Perl will fill the expanded positions with `undef`.

Array Assignment

Assign to individual positions in an array directly by index:

```
my @cats;
$cats[3] = 'Jack';
$cats[2] = 'Tuxedo';
$cats[0] = 'Daisy';
$cats[1] = 'Petunia';
$cats[4] = 'Brad';
$cats[5] = 'Choco';
```

If you assign to an index beyond the array's current bounds, Perl will extend the array for you. As you might expect, all intermediary positions with then contain `undef`. After the first assignment, the array will contain `undef` at positions 0, 1, and 2 and `Jack` at position 3.

As an assignment shortcut, initialize an array from a list:

```
my @cats = ( 'Daisy', 'Petunia', 'Tuxedo', ... );
```

... but remember that these parentheses *do not* create a list. Without parentheses, this would assign `Daisy` as the first and only element of the array, due to operator precedence (*precedence*). `Petunia`, `Tuxedo`, and all of the other cats would be evaluated in void context and Perl would complain. (So would all the other cats, especially Petunia.)

You may assign any expression which produces a list to an array:

```
my @cats     = get_cat_list();
my @timeinfo = localtime();
my @nums     = 1 .. 10;
```

Assigning to a scalar element of an array imposes scalar context, while assigning to the array as a whole imposes list context.

To clear an array, assign an empty list:

```
my @dates = ( 1969, 2001, 2010, 2051, 1787 );
...
@dates    = ();
```

This is one of the only cases where parentheses *do* indicate a list; without something to mark a list, Perl and readers of the code would get confused.

`my @items = ();` is a longer and noisier version of `my @items`. Freshly-declared arrays start out empty. Not "full of `undef`" empty. Really empty.

Array Operations

Sometimes an array is more convenient as an ordered, mutable collection of items than as a mapping of indices to values. Perl provides several operations to manipulate array elements.

The `push` and `pop` operators add and remove elements from the tail of an array, respectively:

```
my @meals;

# what is there to eat?
push @meals, qw( hamburgers pizza lasagna turnip );

# ... but your nephew hates vegetables
pop @meals;
```

You may `push` a list of values onto an array, but you may only `pop` one at a time. `push` returns the new number of elements in the array. `pop` returns the removed element.

Because `push` operates on a list, you can easily append the elements of one multiple arrays with:

```
push @meals, @breakfast, @lunch, @dinner;
```

Similarly, `unshift` and `shift` add elements to and remove an element from the start of an array, respectively:

```
# expand our culinary horizons
unshift @meals, qw( tofu spanakopita taquitos );

# rethink that whole soy idea
shift @meals;
```

`unshift` prepends a list of elements to the start of the array and returns the new number of elements in the array. `shift` removes and returns the first element of the array. Almost no one uses these return values.

The `splice` operator removes and replaces elements from an array given an offset, a length of a list slice, and replacement elements. Both replacing and removing are optional; you may omit either. The `perlfunc` description of `splice` demonstrates its equivalences with `push`, `pop`, `shift`, and `unshift`. One effective use is removal of two elements from an array:

```
my ($winner, $runnerup) = splice @finalists, 0, 2;

# or
my $winner              = shift @finalists;
my $runnerup            = shift @finalists;
```

The `each` operator allows you to iterate over an array by index and value:

```
    while (my ($position, $title) = each @bookshelf) {
        say "#$position: $title";
    }
```

Array Slices

An *array slice* allows you to access elements of an array in list context. Unlike scalar access of an array element, this indexing operation takes a list of zero or more indices and uses the array sigil (@):

```
    my @youngest_cats = @cats[-1, -2];
    my @oldest_cats   = @cats[0 .. 2];
    my @selected_cats = @cats[ @indexes ];
```

Array slices are useful for assignment:

```
    @users[ @replace_indices ] = @replace_users;
```

The only syntactic difference between an array slice of one element and the scalar access of an array element is the leading sigil. The *semantic* difference is greater: an array slice always imposes list context. An array slice evaluated in scalar context will produce a warning:

```
    Scalar value @cats[1] better written as $cats[1]...
```

An array slice imposes list context on the expression used as its index:

```
    # function called in list context
    my @hungry_cats = @cats[ get_cat_indices() ];
```

A slice can contain zero or more elements--including one:

```
    # single-element array slice; I<list> context
    @cats[-1] = get_more_cats();

    # single-element array access; I<scalar> context
    $cats[-1] = get_more_cats();
```

Arrays and Context

In list context, arrays flatten into lists. If you pass multiple arrays to a normal function, they will flatten into a single list:

```
    my @cats = qw( Daisy Petunia Tuxedo Brad Jack Choco );
    my @dogs = qw( Rodney Lucky Rosie );

    take_pets_to_vet( @cats, @dogs );

    sub take_pets_to_vet {
        # BUGGY: do not use!
        my (@cats, @dogs) = @_;
        ...
    }
```

Within the function, @_ will contain nine elements, not two, because list assignment to arrays is *greedy*. An array will consume as many elements from the list as possible. After the assignment, `@cats` will contain *every* argument passed to the function. `@dogs` will be empty, and woe to anyone who treats Rodney as a cat.

This flattening behavior sometimes confuses people who attempt to create nested arrays:

```
    # creates a single array, not an array of arrays
    my @numbers = (1 .. 10, (11 .. 20, (21 .. 30)));
```

... but this code is effectively the same as either:

```
    # parentheses do not create lists
    my @numbers = ( 1 .. 10, 11 .. 20, 21 .. 30 );

    # creates a single array, not an array of arrays
    my @numbers = 1 .. 30;
```

... because, again, parentheses merely group expressions. They do not *create* lists. To avoid this flattening behavior, use array references (*array_references*).

Array Interpolation

Arrays interpolate in strings as lists of the stringification of each item separated by the current value of the magic global `$"`. The default value of this variable is a single space. Its *English.pm* mnemonic is `$LIST_SEPARATOR`. Thus:

```
    my @alphabet = 'a' .. 'z';
    say "[@alphabet]";
    B<[a b c d e f g h i j k l m>
     B<n o p q r s t u v w x y z]>
```

Per Mark Jason Dominus, localize `$"` with a delimiter to improve your debugging:

```
    # what's in this array again?
    local $" = ')(';
    say "(@sweet_treats)";
    B<(pie)(cake)(doughnuts)(cookies)(cinnamon roll)>
```

Hashes

A *hash* is an aggregate data structure which associates string keys with scalar values. Just as the name of a variable corresponds to something which holds a value, so a hash key refers to something which contains a value. Think of a hash like a contact list: use the names of your friends to look up their birthdays. Other languages call hashes *tables*, *associative arrays*, *dictionaries*, and *maps*.

Hashes have two important properties: they store one scalar per unique key and they provide no specific ordering of keys. Keep that latter property in mind. Though it has always been true in Perl, it's very, very true in modern Perl.

Declaring Hashes

Hashes use the `%` sigil. Declare a lexical hash with:

```
    my %favorite_flavors;
```

A hash starts out empty. You could write `my %favorite_flavors = ();`, but that's redundant.

Hashes use the scalar sigil `$` when accessing individual elements and curly braces `{ }` for keyed access:

```
    my %favorite_flavors;
    $favorite_flavors{Gabi}    = 'Dark chocolate raspberry';
    $favorite_flavors{Annette} = 'French vanilla';
```

Assign a list of keys and values to a hash in a single expression:

```
    my %favorite_flavors = (
        'Gabi',    'Dark chocolate raspberry',
        'Annette', 'French vanilla',
    );
```

Hashes store pairs of keys and values. Perl will warn you if you assign an odd number of elements to a hash. Idiomatic Perl often uses the *fat comma* operator (=>) to associate values with keys, as it makes the pairing more visible:

```
    my %favorite_flavors = (
        Gabi    B<< => >> 'Dark chocolate raspberry',
        Annette B<< => >> 'French vanilla',
    );
```

The fat comma operator acts like the regular comma *and* also automatically quotes the previous bareword (*barewords*). The `strict` pragma will not warn about such a bareword--and if you have a function with the same name as a hash key, the fat comma will *not* call the function:

```
    sub name { 'Leonardo' }

    my %address = (
        name => '1123 Fib Place'
    );
```

The key of this hash will be `name` and not `Leonardo`. To call the function, make the function call explicit:

```
    my %address = (
        B<name()> => '1123 Fib Place'
    );
```

Hash assignment occurs in list context. Any function called in a hash assignment will default to list context without an explicit `scalar()` coercion.

You may occasionally see `undef %hash`, but that's a little ugly. Assign an empty list to empty a hash:

```
    %favorite_flavors = ();
```

Hash Indexing

To access an individual hash value, use the *keyed access* syntax:

```
    my $address = $addressesB<{$name}>;
```

In this example, `$name` contains a string which is also a key of the hash. As with accessing an individual element of an array, the hash's sigil has changed from `%` to `$` to indicate keyed access to a scalar value.

You may also use string literals as hash keys. Perl quotes barewords automatically according to the same rules as fat commas:

```
    # auto-quoted
    my $address = $addresses{Victor};

    # needs quoting; not a valid bareword
    my $address = $addresses{B<'>Sue-LinnB<'>};

    # function call needs disambiguation
    my $address = $addresses{get_nameB<()>};
```

Novices often always quote string literal hash keys, but experienced developers elide the quotes whenever possible. If you code this way, you can use the rare presence of quotes to indicate that you're doing something special on purpose.

Even Perl builtins get the autoquoting treatment:

```
    my %addresses = (
        Leonardo => '1123 Fib Place',
        Utako    => 'Cantor Hotel, Room 1',
    );

    sub get_address_from_name {
        return $addresses{B<+>shift};
    }
```

The unary plus (*unary_coercions*) turns what would be a bareword (`shift`) subject to autoquoting rules into an expression. As this implies, you can use an arbitrary expression--not only a function call--as the key of a hash:

```
    # don't actually I<do> this though
    my $address = $addresses{reverse 'odranoeL'};

    # interpolation is fine
    my $address = $addresses{"$first_name $last_name"};

    # so are method calls
    my $address = $addresses{ $user->name };
```

Hash keys can only be strings. Anything that evaluates to a string is an acceptable hash key. Perl will go so far as to coerce (*coercion*) an expression into a string. For example, if you use an object as a hash key, you'll get the stringified version of that object instead of the object itself:

```
    for my $isbn (@isbns) {
        my $book = Book->fetch_by_isbn( $isbn );

        # unlikely to do what you want
        $books{$book} = $book->price;
    }
```

That stringified hash will look something like `Book=HASH(0x222d148)`. `Book` refers to the class name. `HASH` identifies the object as a blessed reference. `0x22d148` is a number used to identify the object (more precisely: it's the location of the data structure representing the hash in memory, so it's neither quite random nor unique).

Hash Key Existence

The `exists` operator returns a boolean value to indicate whether a hash contains the given key:

```
    my %addresses = (
        Leonardo => '1123 Fib Place',
        Utako    => 'Cantor Hotel, Room 1',
    );

    say "Have Leonardo's address" if exists $addresses{Leonardo};
    say "Have Warnie's address"   if exists $addresses{Warnie};
```

Using `exists` instead of accessing the hash key directly avoids two problems. First, it does not check the boolean nature of the hash *value*; a hash key may exist with a value even if that value evaluates to a boolean false (including `undef`):

```
    my %false_key_value = ( 0 => '' );
    ok %false_key_value,
       'hash containing false key & value should evaluate to a true value';
```

Second, `exists` avoids autovivification (*autovivification*) within nested data structures (*nested_data_structures*).

If a hash key exists, its value may be `undef`. Check that with `defined`:

```
    $addresses{Leibniz} = undef;

    say "Gottfried lives at $addresses{Leibniz}"
        if exists  $addresses{Leibniz}
        && defined $addresses{Leibniz};
```

Accessing Hash Keys and Values

Hashes are aggregate variables, but their pairwise nature is unique. Perl allows you to iterate over a hash's keys, its values, or pairs of its keys and values. The `keys` operator produces a list of hash keys:

```
    for my $addressee (keys %addresses) {
        say "Found an address for $addressee!";
    }
```

The `values` operator produces a list of hash values:

```
    for my $address (values %addresses) {
        say "Someone lives at $address";
    }
```

The each operator produces a list of two-element key/value lists:

```
while (my ($addressee, $address) = each %addresses) {
    say "$addressee lives at $address";
}
```

Unlike arrays, hash elements have no obvious ordering. The ordering depends on the internal implementation of the hash, the particular version of Perl you are using, the size of the hash, and a random factor. Even so, the order of hash items is consistent between keys, values, and each. Modifying the hash may change the order, but you can rely on that order if the hash remains the same. However, even if two hashes have the *same* keys and values, you cannot rely on the iteration order between those hashes being the same. They may have been constructed differently or have had elements removed. Since Perl 5.18, even if you build two hashes in the same way, you cannot depend on the same iteration order between them.

Read the previous paragraph again. You'll be glad you did.

Each hash has only a *single* iterator for the each operator. You cannot reliably iterate over a hash with each more than once; if you begin a new iteration while another is in progress, the former will end prematurely and the latter will begin partway through the iteration. Beware not to call any function which may itself try to iterate over the hash with each.

This is rarely a problem, but it's not fun to debug. Reset a hash's iterator with keys or values in void context:

```
# reset hash iterator
keys %addresses;

while (my ($addressee, $address) = each %addresses) {
    ...
}
```

Hash Slices

A *hash slice* is a list of keys or values of a hash indexed in a single operation. To initialize multiple elements of a hash at once:

```
my %cats;
@cats{qw( Jack Brad Mars Grumpy )} = (1) x 4;
```

This is equivalent to the initialization:

```
my %cats = map { $_ => 1 } qw( Jack Brad Mars Grumpy );
```

Note however that the hash slice assignment can also add to the existing contents of the hash.

Hash slices also allow you to retrieve multiple values from a hash in a single operation. As with array slices, the sigil of the hash changes to @ to indicate list context. The use of the curly braces indicates keyed access and makes the fact that you're working with a hash unambiguous:

```
my @buyer_addresses = @addresses{ @buyers };
```

Hash slices make it easy to merge two hashes:

```
my %addresses        = ( ... );
my %canada_addresses = ( ... );

@addresses{ keys %canada_addresses } = values %canada_addresses;
```

This is equivalent to looping over the contents of `%canada_addresses` manually, but is much shorter. Note that this relies on the iteration order of the hash remaining consistent between `keys` and `values`. Perl guarantees this, but only because these operations occur on the same hash with no modifications to that hash between the `keys` and `values` operations.

What if the same key occurs in both hashes? The hash slice approach always *overwrites* existing key/value pairs in `%addresses`. If you want other behavior, looping is more appropriate.

The Empty Hash

An empty hash contains no keys or values. It evaluates to a false value in a boolean context. A hash which contains at least one key/value pair evaluates to a true value in boolean context even if all of the keys or all of the values or both would themselves evaluate to boolean false values.

```
use Test::More;

my %empty;
ok ! %empty, 'empty hash should evaluate false';

my %false_key = ( 0 => 'true value' );
ok %false_key, 'hash containing false key should evaluate to true';

my %false_value = ( 'true key' => 0 );
ok %false_value, 'hash containing false value should evaluate to true';

done_testing();
```

In scalar context, a hash evaluates to a string which represents the ratio of full buckets in the hash--internal details about the hash implementation that you can safely ignore. In a boolean scalar context, this ratio evaluates to a false value, so remember *that* instead of the ratio details.

In list context, a hash evaluates to a list of key/value pairs similar to the list produced by the `each` operator. However, you *cannot* iterate over this list the same way you can iterate over the list produced by `each`. This loop will never terminate:

```
# infinite loop for non-empty hashes
while (my ($key, $value) = %hash) {
    ...
}
```

You *can* loop over the list of keys and values with a `for` loop, but the iterator variable will get a key on one iteration and its value on the next, because Perl will flatten the hash into a single list of interleaved keys and values.

Hash Idioms

Because each key exists only once in a hash, assigning the same key to a hash multiple times stores only the most recent value associated with that key. This behavior has advantages! For example, to find unique elements of a list:

```
my %uniq;
undef @uniq{ @items };
my @uniques = keys %uniq;
```

Using `undef` with a hash slice sets the values of the hash to `undef`. This idiom is the cheapest way to perform set operations with a hash.

Hashes are useful for counting elements, such as IP addresses in a log file:

```
my %ip_addresses;

while (my $line = <$logfile>) {
    chomp $line;
    my ($ip, $resource) = analyze_line( $line );
    $ip_addresses{$ip}++;
    ...
}
```

The initial value of a hash value is `undef`. The postincrement operator (`++`) treats that as zero. This in-place modification of the value increments an existing value for that key. If no value exists for that key, Perl creates a value (`undef`) and immediately increments it to one, as the numification of `undef` produces the value 0.

This strategy provides a useful caching mechanism to store the result of an expensive operation with little overhead:

```
{
    my %user_cache;

    sub fetch_user {
        my $id = shift;
        $user_cache{$id} //= create_user($id);
        return $user_cache{$id};
    }
}
```

This *orcish maneuver* (or-cache) returns the value from the hash, if it exists. Otherwise, it calculates, caches, and returns the value. The defined-or assignment operator (`//=`) evaluates its left operand. If that operand is not defined, the operator assigns to the lvalue the value of its right operand. In other words, if there's no value in the hash for the given key, this function will call `create_user()` with the key and update the hash.

You may see older code which uses the boolean-or assignment operator (`||=`) for this purpose. Remember though that some valid values evaluate as false in a boolean context. The defined-or operator usually makes more sense, as it tests for definedness instead of truthiness.

If your function takes several arguments, use a slurpy hash (*parameter_slurping*) to gather key/value pairs into a single hash as named function arguments:

```
sub make_sundae {
    my %parameters = @_;
    ...
}

make_sundae( flavor  => 'Lemon Burst',
             topping => 'cookie bits' );
```

This approach allows you to set default values:

```
    sub make_sundae {
        my %parameters           = @_;
        B<$parameters{flavor}    //= 'Vanilla';>
        B<$parameters{topping}   //= 'fudge';>
        B<$parameters{sprinkles} //= 100;>
        ...
    }
```

... or include them in the hash initialization, as latter assignments take precedence over earlier assignments:

```
    sub make_sundae {
        my %parameters = (
            B<< flavor    => 'Vanilla', >>
            B<< topping   => 'fudge', >>
            B<< sprinkles => 100, >>
            @_,
        );
        ...
    }
```

Locking Hashes

As hash keys are barewords, they offer little typo protection compared to the function and variable name protection offered by the `strict` pragma. The little-used core module `Hash::Util` can make hashes safer.

To prevent someone from accidentally adding a hash key you did not intend (whether as a typo or from untrusted user input), use the `lock_keys()` function to restrict the hash to its current set of keys. Any attempt to add a new key to the hash will raise an exception. Similarly you can lock or unlock the existing value for a given key in the hash (`lock_value()` and `unlock_value()`) and make or unmake the entire hash read-only with `lock_hash()` and `unlock_hash()`.

This is lax security; anyone can use the appropriate unlocking functions to work around the locking. Yet it does protect against typos and other unintended accidents.

Coercion

Throughout the lifetime of a Perl variable, it may contain values of different types--strings, integers, rational numbers, and more. Instead of attaching type information to variables, Perl relies on the context provided by operators (*value_contexts*) to determine how to handle values. By design, Perl attempts to do what you mean--you may hear this referred to as *DWIM* for *do what I mean* or *dwimmery*.>--though you must be specific about your intentions. If you treat a value as a string, Perl will do its best to *coerce* that value into a string.

Boolean Coercion

Boolean coercion occurs when you test the *truthiness* of a value, such as in an `if` or `while` condition. Numeric 0, `undef`, the empty string, and the string `'0'` all evaluate as false values. All other values--including strings which may be *numerically* equal to zero (such as `'0.0'`, `'0e'`, and `'0` `but true'`)--evaluate as true values.

When a scalar has *both* string and numeric components (*dualvars*), Perl prefers to check the string component for boolean truth. `'0 but true'` evaluates to zero numerically, but it is not an empty string, so it evaluates to a true value in boolean context.

String Coercion

String coercion occurs when using string operators such as comparisons (`eq` and `cmp`), concatenation, `split`, `substr`, and regular expressions, as well as when using a value or an expression as a hash key. The undefined value stringifies to an empty string but produces a "use of uninitialized value" warning. Numbers *stringify* to strings containing their values; the value `10` stringifies to the string `10`. You can even `split` a number into individual digits with:

```
my @digits = split '', 1234567890;
```

Numeric Coercion

Numeric coercion occurs when using numeric comparison operators (such as == and <=>), when performing mathematic operations, and when using a value or expression as an array or list index. The undefined value *numifies* to zero and produces a "Use of uninitialized value" warning. Strings which do not begin with numeric portions numify to zero and produce an "Argument isn't numeric" warning. Strings which begin with characters allowed in numeric literals numify to those values and produce no warnings, such that `10 leptons leaping` numifies to `10` and `6.022e23 moles marauding` numifies to `6.022e23`.

The core module `Scalar::Util` contains a `looks_like_number()` function which uses the same rules as the Perl parser to extract a number from a string.

The strings `Inf` and `Infinity` represent the infinite value and behave as numbers. The string `NaN` represents the concept "not a number". Numifying them produces no "Argument isn't numeric" warning. Beware that Perl's ideas of infinity and not a number may not match your platform's ideas; these notions aren't always portable across operating systems. Perl is consistent even if the rest of the universe isn't.

Reference Coercion

Using a dereferencing operation on a non-reference turns that value *into* a reference. This process of autovivification (*autovivification*) is handy when manipulating nested data structures (*nested_data_structures*):

```
my %users;

$users{Brad}{id} = 228;
$users{Jack}{id} = 229;
```

Although the hash never contained values for `Brad` and `Jack`, Perl helpfully created hash references for them, then assigned each a key/value pair keyed on `id`.

Cached Coercions

Perl's internal representation of values stores both string and numeric values. Stringifying a numeric value does not *replace* the numeric value. Instead, it *adds* a stringified value to the internal representation, which then contains *both* components. Similarly, numifying a string value populates the numeric component while leaving the string component untouched.

Certain Perl operations prefer to use one component of a value over another--boolean checks prefer strings, for example. If a value has a cached representation in a form you do not expect, relying on an implicit conversion may produce surprising results. You almost never need to be explicit about what you expect. Your author can recall doing so twice in two decades. Even so, knowing that this caching occurs may someday help you diagnose an odd situation.

Dualvars

The multi-component nature of Perl values is available to users in the form of *dualvars*. The core module `Scalar::Util` provides a function `dualvar()` which allows you to bypass Perl coercion and manipulate the string and numeric components of a value separately:

```
use Scalar::Util 'dualvar';
my $false_name = dualvar 0, 'Sparkles & Blue';

say 'Boolean true!'  if        !! $false_name;
say 'Numeric false!' unless  0  + $false_name;
say 'String true!'   if     ''  . $false_name;
```

Packages

A Perl *namespace* associates and encapsulates various named entities. It's like your family name or a brand name. Unlike a real-world name, a namespace implies no direct relationship between entities. Such relationships may exist, but they are not required to.

A *package* in Perl is a collection of code in a single namespace. The distinction is subtle: the package represents the source code and the namespace represents the internal data structure Perl uses to collect and group that code.

The `package` builtin declares a package and a namespace:

```
package MyCode;

our @boxes;

sub add_box { ... }
```

All global variables and functions declared or referred to after the package declaration refer to symbols within the `MyCode` namespace. You can refer to the `@boxes` variable from the `main` namespace only by its *fully qualified* name of `@MyCode::boxes`. A fully qualified name includes a complete package name, so that you can call the `add_box()` function by `MyCode::add_box()`.

The scope of a package continues until the next `package` declaration or the end of the file, whichever comes first. You may also provide a block with `package` to delineate the scope of the declaration:

```
package Pinball::Wizard
{
    our $VERSION = 1969;
}
```

The default package is the `main` package. Without a package declaration, the current package is `main`. This rule applies to one-liners, standalone programs, and even *.pm* files.

Besides a name, a package has a version and three implicit methods, `import()` (*importing*), `unimport()`, and `VERSION()`. `VERSION()` returns the package's version. This is a series of numbers contained in a package global named `$VERSION`. By rough convention, versions are a series of dot-separated integers such as `1.23` or `1.1.10`.

Perl includes a stricter syntax for version numbers, as documented in `perldoc version::Internals`. These version numbers must have a leading `v` character and at least three integer components separated by periods:

```
package MyCode v1.2.1;
```

Combined with the block form of a `package` declaration, you can write:

```
package Pinball::Wizard v1969.3.7 { ... }
```

You're more likely to see the older version of this code, written as:

```
package MyCode;

our $VERSION = 1.21;
```

Every package inherits a `VERSION()` method from the `UNIVERSAL` base class. This method returns the value of `$VERSION`:

```
my $version = Some::Plugin->VERSION;
```

If you provide a version number as an argument, this method will throw an exception unless the version of the module is equal to or greater than the argument:

```
# require at least 2.1
Some::Plugin->VERSION( 2.1 );

die "Your plugin $version is too old" unless $version > 2;
```

You may override `VERSION()`, though there are few reasons to do so.

Packages and Namespaces

Every `package` declaration creates a new namespace, if necessary. After Perl parses that declaration, it will store all subsequent package global symbols (global variables and functions) in that namespace.

Perl has *open namespaces*. You can add functions or variables to a namespace at any point, either with a new package declaration:

```
package Pack
{
    sub first_sub { ... }
}

Pack::first_sub();

package Pack
{
    sub second_sub { ... }
}

Pack::second_sub();
```

... or by declaring functions with fully qualified names:

```
# implicit
package main;

sub Pack::third_sub { ... }
```

You can add to a package at any point during compilation or runtime, regardless of the current file, though building up a package from multiple declarations in multiple files can make code difficult to spelunk.

Namespaces can have as many levels as your organizational scheme requires, though namespaces are not hierarchical. The only relationship between separate packages is semantic, not technical. Many projects and businesses create their own top-level namespaces. This reduces the possibility of global conflicts and helps to organize code on disk. For example:

* `StrangeMonkey` is the project name

* `StrangeMonkey::UI` organizes user interface code

* `StrangeMonkey::Persistence` organizes data management code

* `StrangeMonkey::Test` organizes testing code for the project

... and so on. This is a convention, but it's a useful one.

References

Perl usually does what you expect, even if what you expect is subtle. Consider what happens when you pass values to functions:

```
sub reverse_greeting {
    my $name = reverse shift;
    return "Hello, $name!";
}

my $name = 'Chuck';
say reverse_greeting( $name );
say $name;
```

Outside of the function, `$name` contains `Chuck`, even though the value passed into the function gets reversed into kcuhC. You probably expected that. The value of $name outside the function is separate from the $name inside the function. Modifying one has no effect on the other.

Consider the alternative. If you had to make copies of every value before anything could possibly change them out from under you, you'd have to write lots of extra defensive code.

Sometimes it's useful to modify values in place. If you want to pass a hash full of data to a function to modify it, creating and returning a new hash for each change could be tedious and inefficient, at least without some amazing compiler magic.

Perl provides a mechanism by which to refer to a value without making a copy. Any changes made to that *reference* will update the value in place, such that *all* references to that value will refer to the modified value. A reference is a first-class scalar data type which refers to another first-class data type.

Scalar References

The reference operator is the backslash (\). In scalar context, it creates a single reference which refers to another value. In list context, it creates a list of references. To take a reference to $name:

```
my $name     = 'Larry';
my $name_ref = B<\>$name;
```

You must *dereference* a reference to evaluate the value to which it refers. Dereferencing requires you to add an extra sigil for each level of dereferencing:

```
sub reverse_in_place {
    my $name_ref = shift;
    B<$$name_ref>   = reverse B<$$name_ref>;
}

my $name = 'Blabby';
reverse_in_place( B<\>$name );
say $name;
```

The double scalar sigil ($$) dereferences a scalar reference.

Parameters in @_ behave as *aliases* to caller variables (*iteration_and_aliasing*), so you can modify them in place:

```
sub reverse_value_in_place {
    $_[0] = reverse $_[0];
}

my $name = 'allizocohC';
reverse_value_in_place( $name );
say $name;
```

You usually don't want to modify values this way--callers rarely expect it, for example. Assigning parameters to lexicals within your functions makes copies of the values in @_ and avoids this aliasing behavior.Modifying a value in place or returning a reference to a scalar can save memory. Because Perl copies values on assignment, you could end up with multiple copies of a large string. Passing around references means that Perl will only copy the references--a far cheaper operation. Before you modify your code to pass only references, however, measure to see if this will make a difference.

Complex references may require a curly-brace block to disambiguate portions of the expression. You may *always* use this syntax, though sometimes it clarifies and other times it obscures:

```
sub reverse_in_place {
    my $name_ref   = shift;
    B<${ $name_ref }> = reverse B<${ $name_ref }>;
}
```

If you forget to dereference a scalar reference, Perl will likely coerce the reference into a string value of the form `SCALAR(0x93339e8)` or a numeric value such as `0x93339e8`. This value indicates the type of reference (in this case, `SCALAR`) and the location in memory of the reference (because that's an unambiguous design choice, not because you can do anything with the memory location itself).

Perl does not offer native access to memory locations. The address of the reference is a value used as an identifier. Unlike pointers in a language such as C, you cannot modify the address of a reference or treat it as an address into memory. These addresses are *mostly* unique because Perl may reuse storage locations as it reclaims unused memory.

Array References

Array references are useful in several circumstances:

* To pass and return arrays from functions without list flattening

* To create multi-dimensional data structures

* To avoid unnecessary array copying

* To hold anonymous data structures

Use the reference operator to create a reference to a declared array:

```
my @cards     = qw( K Q J 10 9 8 7 6 5 4 3 2 A );
my $cards_ref = B<\>@cards;
```

Any modifications made through `$cards_ref` will modify `@cards` and vice versa. You may access the entire array as a whole with the `@` sigil, whether to flatten the array into a list (list context) or count its elements (scalar context):

```
my $card_count = B<@$cards_ref>;
my @card_copy  = B<@$cards_ref>;
```

Access individual elements with the dereferencing arrow (`->`):

```
my $first_card = B<< $cards_ref->[0]  >>;
my $last_card  = B<< $cards_ref->[-1] >>;
```

The arrow is necessary to distinguish between a scalar named `$cards_ref` and an array named `@cards_ref`. Note the use of the scalar sigil (*sigils*) to access a single element.

An alternate syntax prepends another scalar sigil to the array reference. It's shorter but uglier to write `my $first_card = `**`$$cards_ref[0]`**`;`.

Use the curly-brace dereferencing syntax to slice (*array_slices*) an array reference:

```
my @high_cards = B<@{ $cards_ref }>[0 .. 2, -1];
```

You *may* omit the curly braces, but their grouping often improves readability.

To create an anonymous array, surround a list-producing expression with square brackets:

```
my $suits_ref = [qw( Monkeys Robots Dinos Cheese )];
```

This array reference behaves the same as named array references, except that the anonymous array brackets *always* create a new reference. Taking a reference to a named array in its scope always refers to the *same* array. For example:

```
my @meals      = qw( soup sandwiches pizza );
my $sunday_ref = \@meals;
my $monday_ref = \@meals;

push @meals, 'ice cream sundae';
```

... both `$sunday_ref` and `$monday_ref` now contain a dessert, while:

```
my @meals      = qw( soup sandwiches pizza );
my $sunday_ref = [ @meals ];
my $monday_ref = [ @meals ];

push @meals, 'berry pie';
```

... neither `$sunday_ref` nor `$monday_ref` contains a dessert. Within the square braces used to create the anonymous array, list context flattens the `@meals` array into a list unconnected to `@meals`.

Hash References

Use the reference operator on a named hash to create a *hash reference*:

```
my %colors = (
    blue   => 'azul',
    gold   => 'dorado',
    red    => 'rojo',
    yellow => 'amarillo',
    purple => 'morado',
);

my $colors_ref = B<\%>colors;
```

Access the keys or values of the hash by prepending the reference with the hash sigil %:

```
my @english_colors = keys   B<%$colors_ref>;
my @spanish_colors = values B<%$colors_ref>;
```

Access individual values of the hash (to store, delete, check the existence of, or retrieve) by using the dereferencing arrow or double scalar sigils:

```
sub translate_to_spanish {
        my $color = shift;
        return B<< $colors_ref->{$color} >>;
        # or return B<< $$colors_ref{$color} >>;
    }
```

Use the array sigil (@) and disambiguation braces to slice a hash reference:

```
    my @colors  = qw( red blue green );
    my @colores = B<@{ $colors_ref }{@colors}>;
```

Create anonymous hashes in place with curly braces:

```
    my $food_ref = B<{>
        'birthday cake' => 'la torta de cumpleaE<ntilde>os',
        candy           => 'dulces',
        cupcake         => 'bizcochito',
        'ice cream'     => 'helado',
    B<}>;
```

As with anonymous arrays, anonymous hashes create a new anonymous hash on every execution.

The common novice error of assigning an anonymous hash to a standard hash produces a warning about an odd number of elements in the hash. Use parentheses for a named hash and curly brackets for an anonymous hash.

Function References

Perl supports *first-class functions*; a function is a data type just as is an array or hash. In other words, Perl supports *function references*. This enables many advanced features (*closures*). Create a function reference by using the reference operator and the function sigil (&) on the name of a function:

```
    sub bake_cake { say 'Baking a wonderful cake!' };

    my $cake_ref = B<\&>bake_cake;
```

Without the *function sigil* (&), you will take a reference to the function's return value or values.

Create anonymous functions with the bare `sub` keyword:

```
    my $pie_ref = B<sub { say 'Making a delicious pie!' }>;
```

The use of the `sub` builtin *without* a name compiles the function but does not register it with the current namespace. The only way to access this function is via the reference returned from `sub`. Invoke the function reference with the dereferencing arrow:

```
    $cake_ref->();
    $pie_ref->();
```

An alternate invocation syntax for function references uses the function sigil (&) instead of the dereferencing arrow. Avoid this `&$cupcake_ref` syntax; it has subtle implications for parsing and argument passing.

Think of the empty parentheses as denoting an invocation dereferencing operation in the same way that square brackets indicate an indexed (array) lookup and curly brackets a keyed (hash) lookup. Pass arguments to the function within the parentheses:

```
$bake_something_ref->( 'cupcakes' );
```

You may also use function references as methods with objects (*moose*). This is useful when you've already looked up the method (*reflection*):

```
my $clean = $robot_maid->can( 'cleanup' );
$robot_maid->$clean( $kitchen );
```

Filehandle References

The lexical filehandle form of `open` and `opendir` operate on filehandle references. Internally, these filehandles are objects of the class `IO::File`. You can call methods on them directly:

```
use autodie 'open';

open my $out_fh, '>', 'output_file.txt';
$out_fh->say( 'Have some text!' );
```

Old code might `use IO::Handle;`. Older code may take references to typeglobs:

```
local *FH;
open FH, "> $file" or die "Can't write '$file': $!";
my $fh = B<\*FH>;
```

This idiom predates the lexical filehandles introduced by Perl 5.6 in March 2000. You may still use the reference operator on typeglobs to take references to package-global filehandles such as `STDIN`, `STDOUT`, `STDERR`, or `DATA`--but these are all global names anyhow.

As lexical filehandles respect explicit scoping, they allow you to manage the lifespan of filehandles as a feature of Perl's memory management.

Reference Counts

Perl uses a memory management technique known as *reference counting*. Every Perl value has a counter attached to it, internally. Perl increases this counter every time something takes a reference to the value, whether implicitly or explicitly. Perl decreases that counter every time a reference goes away. When the counter reaches zero, Perl can safely recycle that value. Consider the filehandle opened in this inner scope:

```
say 'file not open';

{
    open my $fh, '>', 'inner_scope.txt';
    $fh->say( 'file open here' );
}

say 'file closed here';
```

Within the inner block in the example, there's one `$fh`. (Multiple lines in the source code mention it, but there's only one variable, the one named `$fh`.) `$fh` is only in scope in the block. Its value never leaves the block. When execution reaches the end of the block, Perl recycles the variable `$fh` and decreases the reference count of the filehandle referred to by `$fh`. The filehandle's reference count reaches zero, so Perl destroys the filehandle to reclaim memory. As part of the process, it calls `close()` on the filehandle implicitly.

You don't have to understand the details of how all of this works. You only need to understand that your actions in taking references and passing them around affect how Perl manages memory (see *circular_references*).

References and Functions

When you use references as arguments to functions, document your intent carefully. Modifying the values of a reference from within a function may surprise the calling code, which never expected anything else to modify its data. To modify the contents of a reference without affecting the reference itself, copy its values to a new variable:

```
my @new_array = @{ $array_ref };
my %new_hash  = %{ $hash_ref  };
```

This is only necessary in a few cases, but explicit cloning helps avoid nasty surprises for the calling code. If you use nested data structures or other complex references, consider the use of the core module `Storable` and its `dclone` (*deep cloning*) function.

Nested Data Structures

Perl's aggregate data types--arrays and hashes--store scalars indexed by integer or string keys. Note the word scalar. If you try to store an array in an array, Perl's automatic list flattening will make everything into a single array:

```
my @counts = qw( eenie miney moe      );
my @ducks  = qw( huey  dewey louie    );
my @game   = qw( duck  duck  grayduck );

my @famous_triplets = (
    @counts, @ducks, @game
);
```

Perl's solution to this is references (*references*), which are special scalars that can refer to other variables (scalars, arrays, and hashes). You Nested data structures in Perl, such as an array of arrays or a hash of hashes, are possible through the use of references. Unfortunately, the reference syntax can be a little bit ugly.

Use the reference operator, `\`, to produce a reference to a named variable:

```
my @famous_triplets = (
    B<\>@counts, B<\>@ducks, B<\>@game
);
```

... or the anonymous reference declaration syntax to avoid the use of named variables:

```
my @famous_triplets = (
    B<[>qw( eenie miney moe   )B<]>,
    B<[>qw( huey  dewey louie )B<]>,
    B<[>qw( duck  duck  goose )B<]>,
);

my %meals = (
    breakfast => B<{> entree => 'eggs',
                   side   => 'hash browns'   B<}>,
    lunch     => B<{> entree => 'panini',
                   side   => 'apple'         B<}>,
    dinner    => B<{> entree => 'steak',
                   side   => 'avocado salad' B<}>,
);
```

Perl allows an optional trailing comma after the last element of a list. This makes it easy to add more elements in the future.

Use Perl's reference syntax to access elements in nested data structures. The sigil denotes the amount of data to retrieve. The dereferencing arrow indicates that the value of one portion of the data structure is a reference:

```
    my $last_nephew = $famous_triplets[1]->[2];
    my $meal_side   = $meals{breakfast}->{side};
```

The only way to access elements in a nested data structure is through references, so the arrow in the previous examples is superfluous. You may omit it for clarity, except for invoking function references:

```
    my $nephew = $famous_triplets[1][2];
    my $meal   = $meals{breakfast}{side};

    $actions{generous}{buy_food}->( $nephew, $meal );
```

Use disambiguation blocks to access components of nested data structures as if they were first-class arrays or hashes:

```
    my $nephew_count   = @{ $famous_triplets[1] };
    my $dinner_courses = keys %{ $meals{dinner} };
```

... or to slice a nested data structure:

```
    my ($entree, $side) = @{ $meals{breakfast} }{ qw( entree side ) };
```

Whitespace helps, but does not entirely eliminate the noise of this construct. Sometimes a temporary variable provides more clarity:

```
    my $meal_ref        = $meals{breakfast};
    my ($entree, $side) = @$meal_ref{qw( entree side )};
```

... or use `for`'s implicit aliasing to avoid the use of an intermediate reference (though note the lack of `my`):

```
    ($entree, $side) = @{ $_ }{qw( entree side )} for $meals{breakfast};
```

`perldoc perldsc`, the data structures cookbook, gives copious examples of how to use Perl's various data structures.

Autovivification

When you attempt to write to a component of a nested data structure, Perl will create the path through the data structure to the destination as necessary:

```
my @aoaoaoa;
$aoaoaoa[0][0][0][0] = 'nested deeply';
```

After the second line of code, this array of arrays of arrays of arrays contains an array reference in an array reference in an array reference in an array reference. Each array reference contains one element.

Similarly, when you ask Perl to treat an undefined value as if it were a hash reference, Perl will turn that undefined value into a hash reference:

```
my %hohoh;
$hohoh{Robot}{Santa} = 'mostly harmful';
```

This behavior is *autovivification*. While it reduces the initialization code of nested data structures, it cannot distinguish between the honest intent to create missing elements in nested data structures or a typo.

You may wonder at the contradiction between taking advantage of autovivification while enabling `strict`ures. Is it more convenient to catch errors which change the behavior of your program at the expense of disabling error checks for a few well-encapsulated symbolic references? Is it more convenient to allow data structures to grow or safer to require a fixed size and an allowed set of keys?

The answers depend on your project. During early development, allow yourself the freedom to experiment. While testing and deploying, consider an increase of strictness to prevent unwanted side effects. The `autovivification` pragma (*pragmas*) from the CPAN lets you disable autovivification in a lexical scope for specific types of operations. Combined with the `strict` pragma, you can enable these behaviors where and as necessary.

You *can* verify your expectations before dereferencing each level of a complex data structure, but the resulting code is often lengthy and tedious. It's better to avoid deeply nested data structures by revising your data model to provide better encapsulation.

Debugging Nested Data Structures

The complexity of Perl's dereferencing syntax combined with the potential for confusion with multiple levels of references can make debugging nested data structures difficult. The core module `Data::Dumper` converts values of arbitrary complexity into strings of Perl code, which helps visualize what you have:

```
use Data::Dumper;

my $complex_structure = {
    numbers => [ 1 .. 3 ];
    letters => [ 'a' .. 'c' ],
    objects => {
        breakfast => $continental,
        lunch     => $late_tea,
        dinner    => $banquet,
    },
};
```

```
print Dumper( $my_complex_structure );
```

... which might produce something like:

```
$VAR1 = {
    'numbers' => [
                  1,
                  2,
                  3
                 ],
    'letters' => [
                  'a',
                  'b',
                  'c'
                 ],
    'meals' => {
        'dinner' => bless({...}, 'Dinner'),
        'lunch' => bless({...}, 'Lunch'),
        'breakfast' => bless({...}, 'Breakfast'),
    },
};
```

Use this when you need to figure out what a data structure contains, what you should access, and what you accessed instead. As the elided example alludes, `Data::Dumper` can dump objects as well as function references (if you set `$Data::Dumper::Deparse` to a true value).

While `Data::Dumper` is a core module and prints Perl code, its output is verbose. Some developers prefer the use of the `YAML::XS` or `JSON` modules for debugging. They do not produce Perl code, but their outputs can be much clearer to read and to understand.

Circular References

Perl's memory management system of reference counting (*reference_counts*) has one drawback. Two references which point to each other, whether directly or indirectly, form a *circular reference* that Perl cannot destroy on its own. Consider a biological model, where each entity has two parents and zero or more children:

```
my $alice  = { mother => '',     father => ''     };
my $robin  = { mother => '',     father => ''     };
my $cianne = { mother => $alice, father => $robin };

push @{ $alice->{children} }, $cianne;
push @{ $robin->{children} }, $cianne;
```

Both `$alice` and `$robin` contain an array reference which contains `$cianne`. Because `$cianne` is a hash reference which contains `$alice` and `$robin`, Perl will never decrease the reference count of any of these three people to zero. It doesn't recognize that these circular references exist, and it can't manage the lifespan of these entities.

Either break the reference count manually yourself (by clearing the children of `$alice` and `$robin` or the parents of `$cianne`), or use *weak references*. A weak reference does not increase the reference count of its referent. Use the core module `Scalar::Util`'s `weaken()` function to weaken a reference:

```
use Scalar::Util 'weaken';

my $alice  = { mother => '',     father => ''     };
my $robin  = { mother => '',     father => ''     };
my $cianne = { mother => $alice, father => $robin };

push @{ $alice->{children} }, $cianne;
push @{ $robin->{children} }, $cianne;

B<< weaken( $cianne->{mother} ); >>
B<< weaken( $cianne->{father} ); >>
```

`$cianne` will retain usable references to `$alice` and `$robin`, but those weak references do not count toward the number of remaining references to the parents. If the reference count of `$alice` reaches zero, Perl's garbage collector will reclaim her record, even though `$cianne` has a weak reference to `$alice`. When `$alice` gets reclaimed, `$cianne`'s reference to `$alice` will be set to `undef`.

Most data structures do not need weak references, but when they're necessary, they're invaluable.

Alternatives to Nested Data Structures

While Perl is content to process data structures nested as deeply as you can imagine, the human cost of understanding these data structures and their relationships--to say nothing of the complex syntax--is high. Beyond two or three levels of nesting, consider whether modeling various components of your system with classes and objects (*moose*) will allow for clearer code.