The Perl Language

The Perl language has several smaller parts which combine to form its syntax. Unlike spoken language, where nuance and tone of voice and intuition allow people to communicate despite slight misunderstandings and fuzzy concepts, 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: variables, functions, packages, classes, and even filehandles have names. These names all start with a letter or an underscore. They may optionally include any combination of letters, numbers, and underscores. When the `utf8` pragma (*pragmas*, *unicode*) is in effect, you may use any valid UTF-8 characters in identifiers. These are all valid Perl identifiers:

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

sub anAwkwardName3;

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

These are invalid Perl identifiers:

```
my $invalid name;
my @3;
my %~flags;

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

These rules only apply to names which appear in literal form in source code; that is, if you've typed it directly like `sub fetch_pie` or `my $wafflemaker`.

Perl's dynamic nature makes it possible to refer to entities with names generated at runtime or provided as input to a program. These are *symbolic lookups*. You get more flexibility this way at the expense of some safety. In particular, invoking functions or methods indirectly or looking up namespaced symbols lets you bypass Perl's parser, which is the only part of Perl that enforces these grammatic rules. Be aware that doing so can produce confusing code; a hash (*hashes*) or nested data structure (*nested_data_structures*) is often clearer.

Variable Names and Sigils

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

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

These sigils offer a sort of namespacing for the variables, where it's possible (though often confusing)

to have variables of the same name but different types:

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

Perl won't get confused, but people reading the code will.

Perl 5 uses *variant sigils*, where the sigil on a variable may change depending on what you do with it. For example, to access a (scalar) element of an array or a hash, the sigil changes to the dollar sign ($) character:

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

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

In the latter two lines, using a scalar element of an aggregate as an *lvalue* (the target of an assignment, on the left side of the = character) imposes scalar context (*context_philosophy*) on the *rvalue* (the value assigned, on the right side of the = character).

Similarly, accessing multiple elements of a hash or an array--an operation known as *slicing*--uses the at symbol (@) as the leading sigil and enforces list context when used as an lvalue:

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

Add an example that uses a hash slice as an lvalue? I.E.

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

The most reliable way to determine the type of a variable--scalar, array, or hash--is to look at the operations performed on it. Scalars support all basic operations, such as string, numeric, and boolean manipulations. Arrays support indexed access through square brackets. Hashes support keyed access through curly brackets.

Package-Qualified Names

Occasionally you may need to refer to functions or variables in a separate namespace. Often you will need to refer to a class by its *fully-qualified name*. These names are collections of package names joined by double colons (`::`). That is, `My::Fine::Package` refers to a logical collection of variables and functions.

While the standard naming rules apply to package names, by convention user-defined packages all start with uppercase letters. The Perl core reserves lowercase package names for built-in pragmas (*pragmas*), such as `strict` and `warnings`. This is a policy enforced by community guidelines, rather than technical mechanisms.

Namespaces do not nest in Perl 5. The relationship between between `Some::Package` and `Some::Package::Refinement` is only a storage mechanism, with no further implications on the relationships between parent and child or sibling packages. When Perl looks up a symbol in `Some::Package::Refinement`, it looks in the `main::` symbol table for a symbol representing the `Some::` namespace, then in there for the `Package::` namespace, and so on. It's your responsibility to make any *logical* relationships between entities obvious when you choose names and organize your code.

Variables

A *variable* in Perl is a storage location for a value (*values*). You can work with values directly, but all but the most trivial code works with variables. A variable is a level of indirection; it's easier to explain the Pythagorean theorem in terms of the variables a, b, and c than with the side lengths of every right triangle you can imagine. This may seem basic and obvious, but to write robust, well-designed, testable, and composable programs, you must identify and exploit points of genericity wherever possible.

Variable Naming

Not all variables require names, but most of the variables you will encounter in your programs will have names. Variables in Perl 5 must conform to the standard rules of identifier naming (*names*). Variables also have leading sigils.

the Variable Naming section seems quite redundant with the Variable Names and Sigils section from names.pod, which gets inserted directly above this section in chapter 3

Variable Scopes

Variables also have visibility, depending on their scope (*scope*). Most of the variables you will encounter have lexical scope *lexical_scope*), as the expected visibility of these variables is local.

Package variables have global visibility throughout the program, but they are only available when prefixed with the name of the containing package. For example:

```
package Store::IceCream;

use vars '$num_flavors';
$num_flavors = 42;
```

The `vars` pragma declares a global variable without violating the rules of the `strict` pragma. Within the `Store::IceCream` package, `$num_flavors` is available without a namespace prefix. Outside of that package, it is available as `$Store::IceCream::num_flavors`. If `$num_flavors` were a lexical variable (declared with `my`), it would not be available outside of the enclosing scope.

You may also declare package variables with the `our` keyword. Unlike `vars` variables, these variables have a lexical scope:

```
{
    package Store::Candy;

    our $jellybeans;
}

# $Store::Candy::jellybeans not visible as $jellybeans here
```

... yet remember that files themselves have their own lexical scopes, where the `package` declaration on its own does not create a new scope:

```
package Store::Toy;
```

```
our $discount = 0.10;

package Store::Music;

# $Store::Toy::discount still visible as $discount
say "Our current discount is $discount!";
```

This difference in visibility between package variables and lexical variables is apparent in the different storage mechanisms of these variables within Perl 5 itself. Lexical variables get stored in *lexical pads* attached to scopes. Every new entry into such a lexical scope requires Perl to create a new pad to contain the values of the variables for that particular entry into the scope. (This is how a function can call itself and not clobber the values of existing variables.)

Package variables have a storage mechanism called symbol tables. Each package has a single symbol table, and every package variable has an entry in this table. You can inspect and modify this symbol table from Perl; this is how importing works (*importing*).

Variable Sigils

In Perl 5, the sigil of the variable in a declaration determines the type of the variable, whether scalar, array, or hash. The sigil of the variable used to access the variable determines the type of access to its value. Sigils on variables vary depending on what you do to the variable. For example, you declare an array as `@values`. You access the first element--a single value--of the array with `$values[0]`. You access a list of values from the array with `@values[ @indices ]`. See the arrays (*arrays*) and hashes (*hashes*) sections for more.

Anonymous Variables

Perl 5 variables do not *need* names; Perl can allocate storage space for variables without storing them in lexical pads or symbol tables. These are *anonymous* variables. The only way to access them is by reference (*references*).

Variables, Types, and Coercion

Perl 5 variables do not enforce types on their values. 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. The types of the *values* is flexible (or dynamic), but the type of the *variable* is static. A scalar variable can only hold scalars. An array variable only contains lists. A hash variable must contain an even-sized list of key/value pairs.

Assigning to a variable may cause coercion (*coercion*). 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 to a scalar imposes scalar context on the operation and produces the number of elements in the array:

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

The relationship between variable types, sigils, and context is vital to a proper understanding of Perl.

Values

Effective Perl programs depend on the accurate representation and manipulation of values.

Computer programs contain *variables*: containers which hold *values*. Values are the actual data the programs manipulate. While it's easy to explain what that data might be--your aunt's name and address, the distance between your office and a golf course on the moon, or the weight of all cookies you've eaten in the past year--the rules regarding the format of that data are often strict. Writing an effective program often means understanding the best (simplest, fastest, most compact, or easiest) way of representing that data.

While the structure of a program depends heavily on the means by which you model your data with appropriate variables, these variables would be meaningless if they couldn't accurately contain the data itself--the values.

Strings

A *string* is a piece of data with no particular formatting, no particular contents, and no semantic meaning beyond the fact that it's a string. It could be your name. It could be the contents of an image file read from your hard drive. It could be the Perl program itself. A string has no meaning to the program until you give it meaning. A string is a fixed amount of data delineated by quotes of some form, yet Perl strings can grow or shrink as you add to or remove from them.

Most strings use either single or double quotes:

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

Characters in a *single-quoted string* represent themselves literally, with two exceptions. You may embed a single quote inside a single-quoted string by escaping the quote with a leading backlash:

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

If a backslash occurs at the end of the string, escape it with another backslash, lest the Perl parser believe you are trying to escape the trailing single quote:

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

Is it worth mentioning \\ here? Like, what happens when you put \ or \\ or \\\ or \\' or \\\' in a single-quoted string?

A *double-quoted string* has more complex (and often, more useful) behavior. For example, you may encode non-printable characters in the string:

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

A string declaration may cross logical newlines, such that these two strings are equivalent:

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

You *can* enter these characters directly in the strings, but it's often difficult to see the visual distinction

between one tab character and four (or two or eight) spaces.

You may also *interpolate* the value of a scalar variable or the values of an array within a double-quoted string directly:

```
my $factoid = "Did you know that B<$name> lives at B<$address>?";
```

You may 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>!\"";
```

If you find that hideously ugly, you may use an alternate *quoting operator*. The q operator performs single quoting, while the qq operator performs double quoting. In each case, you may choose your own delimiter for the string. The character immediately following the operator determines the beginning and end of the string. 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^>Didn't need to escape the single quote!B<^>;
my $complaint = B<q{>It's too early to be awake.B<}>;
```

Even though you can declare a complex string with a series of embedded escape characters, sometimes it's easier to declare a multi-line string on multiple lines. The *heredoc* syntax lets you assign one or more lines of a string with a different syntax:

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

    He looked up.  "Time is never on our side, my child.  Do you see the
irony?
    All they know is change.  Change is the constant on which they all can
    agree.  Whereas we, born out of time to remain perfect and perfectly
    self-aware, can only suffer change if we pursue it.  It is against our
    nature.  We rebel against that change.  Shall we consider them greater
    for it?"
    END_BLURB
```

The `<<'END_BLURB'` syntax has three parts. The double angle-brackets introduce the heredoc. The quotes determine whether the heredoc obeys single-quoted or double-quoted behavior with regard to variable and escape character interpolation. They're optional; the default behavior is double-quoted interpolation. The `END_BLURB` itself is an arbitrary identifier which the Perl 5 parser uses as the ending delimiter.

Be careful; regardless of the indentation of the heredoc declaration itself, the ending delimiter *must* begin in the first column of the program:

```
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
END_INGREDIENTS
}
```

If the identifier begins with whitespace, that same whitespace must be present exactly in the ending delimiter. Even if you do indent the identifier, Perl 5 will *not* remove equivalent whitespace from the start of each line of the heredoc.

You may use a string in other contexts, such as boolean or numeric; its contents will determine the resulting value (*coercion*).

Unicode and Strings

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

Perl 5 strings can represent either of two related but different data types:

Sequences of Unicode characters

The Unicode character set contains characters from the scripts of most languages, and various other symbols. Each character has a *codepoint*, a own unique number which identifies it in the Unicode character set.

Sequences of octets

Binary data is a sequence of 8 bit numbers, each of which can represent a number between 0 and 255.

Unicode strings and binary strings look very similar. They each have a `length()`, and you can perform standard string operations on them such as concatenation, splicing, and regular expression processing. Any string which is not purely binary data is textual data, and 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. Any additional meaning of the string's contents are your responsibility.

Character Encodings

A Unicode string is a sequence of octets which represent 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. Others, represent a subset of Unicode characters. For example, ASCII encodes plain English text with no accented characters and Latin-1 can represent text in most languages which use the Latin script.

If you always decode to and from the appropriate encoding at the inputs and outputs of your program, you will avoid many problems.

Unicode in Your Filehandles

One source of Unicode input is filehandles. If you tell Perl that a specific filehandle works with encoded text, Perl can convert the data to Unicode strings automatically. To do this, add a IO layer to the mode of the `open` builtin. An *IO layer* wraps around input or output and performs some sort of conversion of the data. In this case, the `:utf8` layer performs decoding of UTF-8 data:

```
use autodie;

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

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

You may also modify an existing filehandle with `binmode`:

```
binmode $fh, ':utf8';
```

This works for input as well as output:

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

Without the `utf8` mode, printing Unicode strings to a filehandle will result in a warning (`Wide character in %s`), because files contain octets, not Unicode characters.

Unicode in Your Data

The core module `Encode` provides a function named `decode()` to convert a scalar containing data in a known format to a Unicode string. For example, if you have UTF-8 data:

```
my $string = decode('utf8', $data);
```

The corresponding `encode()` function converts from Perl's internal encoding to the desired output encoding:

```
my $latin1 = encode('iso-8859-1', $string);
```

Unicode in Your Programs

You may include Unicode characters in your programs in three ways. The easiest is to use the `utf8` pragma (*pragmas*), which tells the Perl parser to interpret the rest of the source code file with the UTF-8 encoding This allows you to use Unicode characters in strings as well in identifiers:

```
use utf8;

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

my $pounds = 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.

You may also use the Unicode escape sequence to represent character encodings. The syntax `\x{}` represents a single character; place the hex form of the character's Unicode number within the curly brackets:

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

Note that these escapes interpolate only within double-quoted strings.

Some Unicode characters have names. Though these are more verbose, they can be clearer to read than Unicode numbers. You must use the `charnames` pragma to enable them. Use the `\N{}` escape to refer to them:

```
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.

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

This asymmetry can lead to Unicode strings encoded as UTF-8 for output and decoded as Latin-1 when input.

Worse yet, when the text contains only English characters with no accents, the bug hides--because both encodings have the same representation for every such character.

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

If `$name` contains an English name such as `Alice` you will never notice any problem, because the Latin-1 representation is the same as the UTF-8 representation.

If, on the other hand, `$name` contains a name like `José`, `$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:

ASCII string literal

```
my $hello = "Hello, ";
```

The string literal contains octets.

Latin-1 string literal with no explicit encoding, such as:

```
my $hello = "E<iexcl>Hola, ";
```

The string literal contains octets.

Non ASCII string literal with the `utf8` or `encoding` pragma:

```
use utf8;
```

```
my $hello = "E<#x5E9>E<#x5DC>E<#x5D5>E<#x5DD>, ";
```

The string literal contains Unicode characters.

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

If both strings are octet streams, Perl will concatenate them into a new octet string. 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, the concatenation append UTF-8 data to Latin-1 data, producing a sequence of octets which 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 will be incorrectly decoded into five Unicode characters `JosÃ©` instead of `José` because the UTF-8 data means something else when decoded as Latin-1 data.

See `perldoc perluniintro` for far more detail Unicode, encodings, and how to manage incoming and outgoing data in a Unicode world.

Numbers

Perl also supports numbers, both integers and floating-point values. They support scientific notation as well as binary, octal, and hexadecimal representations:

```
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;
```

The emboldened characters are the numeric prefixes for binary, octal, and hex notation respectively. Be aware that the leading zero always indicates octal mode; this can occasionally produce unanticipated confusion.

Even though you can represent floating-point values explicitly in Perl 5 with perfect accuracy, Perl 5 stores them internally in a binary format. Comparing floating-point values is sometimes imprecise in specific ways; consult `perldoc perlnumber` for more details.

You may not use commas to separate thousands in numeric literals because the parser will interpret the commas as comma operators. You *can* use underscores in other places within the number, however. The parser will treat them as invisible characters. Your readers may not. These are equivalent:

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

Consider the most readable alternative, however.

Because of coercion (*coercion*), Perl programmers rarely have to worry about converting text read from outside the program to numbers. Perl will treat anything which looks like a number as a number in numeric contexts. Even though it almost always does so correctly, occasionally it's useful to know if something really does look like a number. The core module `Scalar::Util` contains a function

named `looks_like_number` which returns a true value if Perl will consider the given argument numeric.

The `Regexp::Common` module from the CPAN also provides several well-tested regular expressions to identify valid *types* (whole number, integer, floating-point value) of numeric values.

Undef

Perl 5 has a value which represents an unassigned, undefined, and unknown value: `undef`. Declared but undefined scalar variables contain `undef`:

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

`undef` evaluates to false in boolean context. Interpolating `undef` into a string--or evaluating it in a string context--produces an `uninitialized value` warning:

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

... produces:

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

The Empty List

When used on the right-hand side of an assignment, the `()` construct represents an empty list. When evaluated in scalar context, this evaluates to `undef`. In list context, it is effectively an empty list.

When used on the left-hand side of an assignment, the `()` construct enforces list context. To count the number of elements returned from an expression in list context without using a temporary variable, you use the idiom (*idioms*):

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

Because of the right associativity (*associativity*) of the assignment operator, Perl first evaluates the second assignment by calling `get_all_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_all_clown_hats()`.

You don't have to understand all of the implications of this code right now, but it does demonstrate how a few of Perl's fundamental design features can combine to produce interesting and useful behavior.

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 = (1, 1, 2, 3, 5, 8, 13, 21);
```

... as targets of assignments:

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

... or as lists of expressions:

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

Note that you do not need parentheses to *create* lists; where present, the parentheses in these examples group expressions to change the *precedence* of those expressions (*precedence*).

You may use the range operator to create lists of literals in a compact form:

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

... and you may use the `qw()` operator to split a literal string on whitespace to produce a list of strings:

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

Perl will produce a warning if a `qw()` contains a comma or the comment character (#), because not only are such characters rarely included in a `qw()`, their presence usually indicates an oversight.

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 and 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 is always occurs in list context. Indexing into an array can occur in scalar context (for a single element) or list context (for a slice):

```
# you do not need to understand this all
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()];
```

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';
```

Most programs need more complex control flow. Perl's *control flow directives* change the order of execution--what happens next in the program--depending on the values of arbitrarily complex expressions.

Branching Directives

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

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

This postfix form is useful for simple expressions. A block form groups multiple expressions into a single unit:

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

Note that the postfix form does not *require* parentheses around its condition, while the block form does. The conditional expression may also be complex:

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

... though in this case, parenthesization of the expression in postfix form may add clarityIt may also argue against using the postfix form.:

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

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

```
say "You're no Bob!" unless $name eq 'Bob';
```

Like `if`, `unless` also has a block form. Unlike `if`, the block form of `unless` is much rarer than its postfix form:

```
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
{
    return unless @_;

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

`unless` can be difficult to read with multiple conditions; this is one reason it appears rarely in its block form.

The block forms of `if` and `unless` both work with the `else` directive, which provides code to run when the conditional expression does not evaluate to true (for `if`) or false (for `unless`):

```
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:

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

If you read the previous example out loud, you may notice the awkward pseudocode phrasing: "Unless this name is Bob, do this. Otherwise, do something else." The implied double negative can be confusing. Perl provides both `if` and `unless` to allow you to phrase your conditionals in the most natural and readable way. Likewise, you can choose between positive and negative assertions with regard to the comparison operators you use:

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

The double negative implied by the presence of the `else` block argues against this particular phrasing.

One or more `elsif` directives may follow an `if` block form and may precede any single `else`. You may use as many `elsif` blocks as you like, but you may not change the order in which the block types appear:

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

You may also use the `elsif` block with an `unless` chain, but the resulting code may be unclear. There is no `elseunless`.

There is no `else if` construct; this is a syntax error:

```
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 offers an alternate approach to control flow. It evaluates a conditional expression and evaluates to one of two different results:

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

The conditional expression precedes the question mark character (?) and the colon character (`:`) separates the alternatives. The alternatives are literals or (parenthesized) expressions of arbitrary complexity, including other ternary conditional expressions, though readability may suffer.

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

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

Again, weigh the benefits of clarity versus the benefits of conciseness.

Perl performs a type of behavior known as *short-circuiting* when it encounters complex expressions--expressions composed of multiple evaluated expressions. If 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, 'first subexpression')
                 && ok(1, 'second subexpression');
```

The return value of `ok()` (*testing*) is the boolean value obtained by evaluating the first argument.

This example prints:

```
ok 1 - first subexpression
ok 2 - first subexpression
Both true!
```

When the first subexpression--the first call to `ok`--evaluates to true, Perl must evaluate the second subexpression. When the first subexpression evaluates to false, the entire expression cannot succeed, and there is no need to check subsequent subexpressions:

```
say "Both true!" if ok(0, 'first subexpression')
                 && ok(1, 'second subexpression');
```

This example prints:

```
not ok 1 - first subexpression
```

Even though the second subexpression would obviously succeed, Perl never evaluates it. The logic is similar for a complex conditional expression where either subexpression must be true for the conditional as a whole to succeed:

```
say "Either true!" if ok(1, 'first subexpression')
                   || ok(1, 'second subexpression');
```

This example prints:

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

Again, with 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:

```
if (exists $barbeque{pork} and $barbeque{pork} eq 'shoulder') { ... }
```

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. Empty hashes and arrays evaluate to false.

Perl 5 has no single true value, nor a single false value. Any number that evaluates to 0 is false. This includes `0`, `0.0`, `0e0`, `0x0`, and so on. The empty string (`''`) and `"0"` evaluate to false, but the strings `"0.0"`, `"0e0"`, and so on do not. The idiom `"0 but true"` evaluates to 0 in numeric context but evaluates to true, thanks to its string contents. Both the empty list and `undef` evaluate to false. Empty arrays and hashes return the number 0 in scalar context, so they evaluate to false 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 true in boolean context.

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

Looping Directives

Perl also provides several directives for looping and iteration.

`foreach` `for looping directives; for looping directives; foreach`

The *foreach* style loop evaluates an expression which produces a list and executes a statement or block until it has consumed that list:

```
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 (`$_`) to each in turn. Perl executes the block for each integer and prints the squares of the integers.

Though this is a *foreach*-style loop, Perl treats the keywords `foreach` and `for` interchangeably. As only the *type* of the loop governs its behavior, there are no drawbacks to using the shorter `for` keyword.

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

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

Similar suggestions apply for clarity and complexity.

You may provide a variable to which to assign the values of the expression in place of the topic variable:

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

If you do so, Perl will not set the topic variable (`$_`) to the iterated values. Note also that the scope of the variable `$i` is only valid *within* the loop. If you have declared a lexical `$i` in an outer scope, that value will remain outside the loop:

```
my $i = 'cow';

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

```
    is( $i, 'cow', 'Lexical variable not overwritten in outer scope' );
```

This localization occurs even if you do not redeclare the iteration variable as a lexical:

```
    my $i = 'horse';

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

    is( $i, 'horse', 'Lexical variable still not overwritten in outer
scope' );
```

Iteration and Aliasing

The `for` loop performs *aliasing* of the iterator variable to the values in the iteration such that you can modify values in place during iteration:

```
    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 `foreach` 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:

```
    lc for qw( Huey Dewey Louie );
```

... as this will throw an exception about modification of read-only values. There's little point in doing so anyhow.

You may occasionally see the use of `for` with a single scalar variable to alias `$_` to the variable:

```
    for ($user_input)
    {
        s/(\w)/\\$1/g;  # escape non-word characters
```

```
    s/^\s*|\s$/g;   # trim whitespace
}
```

Iteration and Scoping

Iterator scoping with the topic variable provides one common source of confusion. In this case, `some_function()` modifies `$_` on purpose. If `some_function()` called other code which modified `$_` without explicitly localizing `$_`, the iterated value in `@values` would change. Debugging this can be troublesome:

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

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

If you *must* use `$_` rather than a named variable, lexicalize the topic variable with `my $_`:

```
sub some_function_called_later
{
    # was $_ = shift;
    B<my> $_ = shift;

    s/foo/bar/;
    s/baz/quux/;

    return $_;
}
```

Using a named iteration variable also prevents undesired aliasing behavior through `$_`.

The C-Style For Loop

The C-style *for loop* allows the programmer to manage iteration manually:

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

You must assign to an iteration variable manually, as there is no default assignment to the topic variable. Consequently there is no aliasing behavior either. Though the scope of any declared lexical variable is to the body of the block, a variable *not* declared explicitly in the iteration control section of this construct *will* overwrite its contents:

```
my $i = 'pig';

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

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

This loop has three subexpressions in its looping construct. The first subexpression is an initialization section. It executes once, before the first execution of the loop body. The second subexpression is the conditional comparison subexpression. Perl evaluates this subexpression before each iteration of the loop body. When the subexpression evaluates to a true value, the loop iteration proceeds. When the subexpression evaluates to a false value, the loop iteration stops. The final subexpression executes after each iteration of the loop body.

This may be more obvious with an example:

```
# declared outside to avoid declaration in conditional
my $i;

for (
    # loop initialization subexpression
    say 'Initializing' and $i = 0;

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

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

Note the lack of a trailing semicolon at the iteration ending subexpression as well as the use of the low-precedence `and`; this syntax is surprisingly finicky. When possible, prefer the `foreach` style loop to the `for` loop.

All three subexpressions are optional. You may write an infinite loop with:

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

While and Until

A *while* loop continues until the loop conditional expression evaluates to a boolean false value. An infinite loop is much clearer when written:

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

The means of evaluating the end of iteration condition in a `while` loop differs from a `foreach` loop in that the evaluation of the expression itself does not produce any side effects. If `@values` has one or more elements, this code is also an infinite loop:

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

To prevent such an infinite `while` loop, you must perform a *destructive update* of the `@values` array by modifying the array with each loop iteration:

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

The *until* loop performs the opposite test as the `while` loop. Iteration continues while the loop conditional expression evaluates to false:

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

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

```
use autodie;

open my $fh, '<', $file;

while (<$fh>)
{
    ...
}
```

Perl 5 interprets this `while` loop as if you had written:

```
while (defined($_ = <$fh>))
{
    ...
}
```

Without the implicit `defined`, any line read from the filehandle which evaluated to false 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 finished reading lines from the file.

One common mistake is to forget to remove the line-ending characters from each line; use the `chomp` keyword to do so.

Both `while` and `until` have postfix forms. The simplest infinite loop in Perl 5 is:

```
1 while 1;
```

Any single expression is suitable for a postfix `while` or `until`, such as the classic "Hello, world!" example from 8-bit computers of the early 1980s:

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

Infinite loops may seem silly, but they're actually quite useful. A simple event loop for a GUI program or network server may be:

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

For more complex expressions, use a `do` block:

```
do
```

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

For the purposes of parsing, a `do` block is itself a single expression, though it can 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 no less powerful.

Loops within Loops

You may nest loops within other loops:

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

In this case, explicitly declaring named variables is essential to maintainability. The potential for confusion as to the scoping of iterator variables is too great when using the topic variable.

A common mistake with nesting `foreach` and `while` loops is that it is easy to exhaust a filehandle with a `while` loop:

```
use autodie;

open my $fh, '<', $some_file;

for my $prefix (@prefixes)
{
    # DO NOT USE; likely buggy code
    while (<$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 and will not execute. To solve this problem, you may re-open the file inside the `for` loop (simple to understand, but not always a good use of system resources), slurp the entire file into memory (which may not work if the file is large), or `seek` the filehandle back to the beginning of the file for each iteration (an often overlooked option):

```
use autodie;

open my $fh, '<', $some_file;

for my $prefix (@prefixes)
{
    while (<$fh>)
```

```
    {
        say $prefix, $_;
    }

    seek $fh, 0, 0;
}
```

Loop Control

Sometimes you need to break out of a loop before you have exhausted the iteration conditions. Perl 5'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 all you need to in the current iteration. To loop over lines in a file but skip everything that looks like a comment, one which starts with the character #, you might write:

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

The *last* statement ends the loop immediately. To finish processing a file once you've seen the ending delimiter, you might 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. For example, you could implement a silly file parser that joins lines which end with a backslash with:

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

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

    ...
}
```

... though that's a contrived example.

Nested loops can make the use of these loop control statements ambiguous. In those cases, a *loop label* can disambiguate:

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

    INNER:
    for my $prefix (@prefixes)
    {
        next OUTER unless $prefix;
        say "$prefix: $_";
    }
}
```

If you find yourself nesting loops such that you need labels to manage control flow, consider simplifying your code: perhaps extracting inner loops into functions for clarity.

Continue

The `continue` construct behaves like the third subexpression of a `for` loop; Perl executes its block for each iteration of the loop, even when you exit an iteration with `next`. You may use it with may be used with a `while`, `until`, `with`, or `for` loop. Examples of `continue` are rare, but it's useful any time you want to guarantee that something occurs for every iteration of the loop regardless of how that iteration ends:

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

Given/When

The `given` construct is a feature new to Perl 5.10. It assigns the value of an expression to the topic variable and introduces a block:

```
given ($name)
{
    ...
}
```

Unlike `for`, it does not iterate over an aggregate. It evaluates its value in scalar context, and always assigns to the topic variable:

```
given (my $username = find_user())
{
```

```
        is( $username, $_, 'topic assignment happens automatically' );
    }
```

`given` also lexicalizes the topic variable to prevent accidental modification:

```
    given ('mouse')
    {
        say;
        mouse_to_man( $_ );
        say;
    }

    sub mouse_to_man
    {
        $_ = shift;
        s/mouse/man/;
    }
```

By itself, this feature may seem less than useful. In combination with `when`, it is very useful. Use `given` to *topicalize* a value. Within the associated block, multiple `when` statements match the topic against expressions using *smart-match* semantics. Thus you might code a Rock, Paper, Scissors game:

This seems like an rather verbose example. Also, I'm not sure how to work mention of continue into it. Perhaps something like this (pretty much ripped from perldoc perlsyn) might suffice?

given($foo) { when (undef) { say '$foo is undefined'; } when ([2, 3, 5, 7, 11, 13, 17, 19]) { say '$foo is a prime less than 20'; continue; } when ($_ % 2){ say '$foo is odd'; continue; } when ($_ < 20) { say '$foo is numerically less than 20'; } default { say q{I don't know what to say about $foo}; } }

```
    my @options = ( \&rock, \&paper, \&scissors );

    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.  ";

        given (shift)
        {
            when (/paper/)    { say 'You win!' };
            when (/rock/)     { say 'We tie!'  };
            when (/scissors/) { say 'I win!'   };
            default           { say "I don't understand your move" };
        }
    }

    sub paper
```

```
{
    print "I chose paper.  ";

    given (shift)
    {
        when (/paper/)    { say 'We tie!'  };
        when (/rock/)     { say 'I win!'   };
        when (/scissors/) { say 'You win!' };
        default           { say "I don't understand your move" };
    }
}

sub scissors
{
    print "I chose scissors.  ";

    given (shift)
    {
        when (/paper/)    { say 'I win!'   };
        when (/rock/)     { say 'You win!' };
        when (/scissors/) { say 'We tie!'  };
        default           { say "I don't understand your move" };
    }
}
```

Perl executes the `default` rule when none of the other conditions match.

The CPAN module `MooseX::MultiMethods` demonstrates another technique which reduces this code further.

The `when` construct is even more powerful; it can match against many other types of expressions including scalars, aggregates, references, arbitrary comparison expressions, and even code references. The "Smart matching in detail" section of the *perlsyn* perldoc provides a useful table of all possible match types.

Link to smart match.

Tailcalls

A *tailcall* occurs when the last expression within a function is a call to another function--the return value of the outer function is the return value of the inner function:

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

    return greet_person( $name );
}
```

In this circumstance, returning from `greet_person()` directly to the caller of `log_and_greet_person()` is more efficient than returning to `log_and_greet_person()` and immediately returning *from* `log_and_greet_person()`. Returning directly from `greet_person()` to the caller of `log_and_greet_person()` is an optimization known as *tailcall optimization*.

Perl 5 cannot perform this optimization automatically, but you can perform it manually.

Why would you want to do this? Heavily recursive code (*recursion*), especially mutually recursive code, can quickly consume a lot of memory. Reducing the memory needed for internal bookkeeping of control flow can make otherwise expensive algorithms tractable.

Check Ruslan Zakirov's tailcall module for maturity and applicability here; it won't be core, but it may be worth mentioning. There are two other sections which mention tailcalls; which has primacy?

Scalars

Perl 5's fundamental data type is the *scalar*, which represents 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. Scalar values and scalar context have a deep connection; assigning to a scalar provides scalar context.

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 the *names* guidelines. Scalar variables always use the leading dollar-sign ($) sigil (*sigils*).

The converse is not *universally* true; the scalar sigil applied to an operation on an aggregate variable determines the amount type accessed through that operation (*arrays*, *hashes*).

Scalars and Types

Perl 5 scalars do not have static typing. A scalar variable can contain any type of scalar value without special conversions or casts, and the type of value in a variable can change. This code is legal:

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

Yet even though this is *legal*, it can be confusing. Choose descriptive and unique names for your variables to avoid this confusion.

The type context of evaluation of a scalar may cause Perl to perform a coercion of the value of that scalar (*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        = 97006;
my $city_state_zip  = 'Beaverton, Oregon' . ' ' . $zip_code;
```

You may also perform mathematical operations on strings:

```
my $call_sign = 'KB7MIU';
my $next_sign = $call_sign++;

# also fine as
$next_sign    = ++$call_sign;

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

Note that this magical string increment behavior does not have a corresponding magical decrement behavior. You can't get 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. Note that ZZ9 becomes ZZ0 and ZZ09 becomes ZZ10, however--numbers wrap around but do not carry to alphabetic components.

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 the string or numeric 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.

All of these coercions and operations are possible because Perl 5 scalars can contain numeric parts as well as string parts. The internal data structure which represents a scalar in Perl 5 has a numeric slot and a string slot. Accessing a string in a numeric context eventually produces a scalar with both string and numeric values. The `dualvar()` function within the core `Scalar::Util` module allows you to manipulate both values directly within a single scalar. Similarly, the module's `looks_like_number()` function returns true if the scalar value provided is something Perl 5 would interpret as a number.

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

Be careful that the *strings* `'0.0'` and `'0e0'` are true; this is one place where Perl 5 makes distinction what looks like a number and what really is a number.

One other value is always false: `undef`. This is the value of uninitialized variables as well as a value in its own right.

Arrays

Perl 5 *arrays* are data structures which store zero or more scalars. They're *first-class* data structures, which means that Perl 5 provides a separate data type at the language level. Arrays support indexed access; that is, you can access individual members of the array by integer indexes.

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

```
my @items;
```

Array Elements

Accessing an individual element of an array in Perl 5 requires the scalar sigil. Perl 5 (and you) can recognize that `$cats[0]` refers to the `@cats` array even despite the change of sigil because the square brackets (`[]`) always identify indexed access to an aggregate variable. In simpler terms, that means "look up one thing in a group of things by an integer".

The first element of an array is at index zero:

```
# @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 contained 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;
```

If you need the specific index of the final element of an array, subtract one from the number of elements of the array (because array indexes start at 0):

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

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

You can also use the special variable form of the array to find the last index; replace the @ array sigil with the slightly more unwieldy $#:

```
my $first_index = 0;
B<my $last_index  = $#cats;>

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

That may not read as nicely, however. Most of the time you don't need that syntax, as you can use negative offsets to access an array from the end instead of the start. The final element of an array is available at the index `-1`. The penultimate element of the array is available at index `-2`, and so on. For example:

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

You can resize an array by assigning to $#. If you shrink an array, Perl will discard values which do not fit in the resized array. If you expand an array, Perl will fill in the expanded values with `undef`.

Array Assignment

You can assign to individual positions in an array directly by index:

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

Perl 5 arrays are mutable. They do not have a static size; they expand or contract as necessary.

You don't have to assign in order, either. If you assign to an index beyond where you've assigned before, Perl will extend the array to account for the new size and will fill in all intermediary slots with `undef`.

Assignment in multiple lines can be tedious. You can initialize an array from a list in one step:

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

Remember that the 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*).

Any expression which produces a list in list context can assign to an array:

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

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

To clear an array, assign an empty list:

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

As freshly-declared arrays start out empty, `my @items = ();` is a longer version of `my @items`. Prefer the latter.

Array Slices

You can also access elements of an array in list context with a construct known as an *array slice*. Unlike scalar access of an array element, this indexing operation takes a list of indices and uses the array sigil (@):

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

You can assign to an array slice as well:

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

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

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

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

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 enforces list context. Any array slice evaluated in scalar context will produce a warning:

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

An array slice imposes list context (*context_philosophy*) on the expression used as its index:

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

Array Operations

Managing array indices can be a hassle. Because Perl 5 can expand or contract arrays as necessary, the language also provides several operations to treat arrays as stacks, queues, and the like.

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

```
my @meals;

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

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

You may `push` as many elements as you like onto an array. Its second argument is a list of values. You may only `pop` one argument at a time. `push` returns the updated number of elements in the array. `pop` returns the removed element.

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

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

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

`unshift` prepends a list of zero or more 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.

Few programs use the return values of `push` and `unshift`. Writing this chapter led to a patch to Perl 5 to optimize the use of `push` in void context.

`splice` is another important--if less frequently used--array operator. It removes and replaces elements from an array given an offset, a length of a list slice, and replacements. Both replacing and removing are optional; you may omit either behavior. The `perlfunc` description of `splice` demonstrates its equivalences with `push`, `pop`, `shift`, and `unshift`. Note that the other operators are shorter and simpler to read and understand.

Arrays often contain elements to process in a loop; see *looping_directives* for more detail about Perl 5 control flow and array processing.

Arrays and Context

In list context, arrays flatten into lists. If you pass multiple arrays to a normal Perl 5 function, they will flatten into a single list:

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

take_pets_to_vet( @cats, @dogs );
```

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

Within the function, `@_` will contain seven elements, not two.

As well, 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.

This flattening behavior sometimes confuses novices who attempt to create nested arrays in Perl 5:

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

While some people may initially expect this code to produce an array where the first ten elements are the numbers one through ten and the eleventh element is an array containing the numbers eleven through 20 and an array containing the numbers twenty-one through thirty, this code instead produces an array containing the numbers one through 30, inclusive. Remember that parentheses do not *create* lists in these circumstances--they only group expressions.

The solution to this flattening behavior is the same for passing arrays to functions and for creating nested arrays (*references*).

Array Interpolation

Arrays interpolate in double quoted strings as a list 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 n o p q r s t u v w x y z]>
```

Temporarily localizing and assigning another value to `$"` for debugging purposes is very handyDue credit goes to Mark-Jason Dominus for demonstrating this example several years ago:

```
# what's in this array again?
{
    local $" = ')(';
    say "(@sweet_treats)";
}
```

... which produces the result:

```
(pie)(cake)(doughnuts)(cookies)(raisin bread)
```

Hashes

A *hash* is a first-class Perl data structure which associates string keys with scalar values. You might have encountered them as *tables*, *associative arrays*, *dictionaries*, or *maps* in other programming languages. In the same way that the name of a variable corresponds to a storage location, a key in a hash refers to a value.

A well-respected, if hoary, analogy is to think of a hash like you would a telephone book: use your friend's name to look up her number.

Hashes have two important properties. First, they store one scalar per unique key. Second, they do not provide any specific ordering of keys. A hash is a big container full of key/value pairs.

Declaring Hashes

A hash has the % sigil. Declare a lexical hash with:

```
my %favorite_flavors;
```

A hash starts out empty, with no keys or values. In boolean context, a hash returns false if it contains no keys. Otherwise, it returns a string which evaluates to true.

You can assign and access individual elements of a hash:

```
my %favorite_flavors;
$favorite_flavors{Gabi}    = 'Mint chocolate chip';
$favorite_flavors{Annette} = 'French vanilla';
```

Hashes use the scalar sigil $ when accessing individual elements and curly braces `{ }` for string indexing.

You may assign a list of keys and values to a hash in a single expression:

```
my %favorite_flavors = (
    'Gabi',    'Mint chocolate chip',
    'Annette', 'French vanilla',
);
```

If you assign an odd number of elements to the hash, you will receive a warning that the results are not what you anticipated. It's often more obvious to use the *fat comma* operator to associate values with keys, as it makes the required pairing more visible:

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

The fat comma operator acts like the regular comma, but it also causes the Perl parser to treat the previous bareword (*barewords*) as if it were a quoted word. The `strict` pragma will not warn about the bare word, 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 the hash will be `name` and not `Leonardo`. If you intend to call the function to get the key, disambiguate your intent by making the call explicit:

```
my %address =
(
```

```
    B<name()> => '1123 Fib Place',
);
```

Hash assignment occurs in list context; if you call a function within this assignment, you may have to use `scalar()` to disambiguate its context.

To empty a hash, assign to it an empty listUnary `undef` also works, but it's somewhat more rare.:

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

Hash Indexing

Because a hash is an aggregate, you can access individual values with an indexing operation. Use a key as an index (a *keyed access* operation) to retrieve a value from a hash:

```
my $address = $addresses{$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<()>};
```

You might find it clearer always to quote string literal hash keys, but the autoquoting behavior is so well established in Perl 5 culture that it's better to reserve the quotes for extraordinary circumstances, where they broadcast your intention to do something different.

Even Perl 5 keywords 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*; *numeric_operators*) 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() };
```

Anything that evaluates to a string is an acceptable hash key. Of course, hash keys can only be strings. 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;
}
```

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`). Second, with nested data structures, it avoids autovivifying (*autovivification*) the value.

The corresponding operator for hash values is `defined`. 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 they behave slightly differently from arrays. In particular, you can iterate over the keys of a hash, the values, of a hash, or pairs of keys and values. The `keys` operator returns a list of keys of the hash:

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

```
}
```

You may assign a numeric value to `keys` to modify the expected size of the hash. This is a minor optimization you almost never need.

The `values` operator returns a list of values of the hash:

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

The `each` operator returns a list of two-element lists of the key and the value:

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

The `while` loop is important to the `each` example, as you cannot declare multiple lexical loop iterator variables in a `for` loop.

Unlike arrays, there is no obvious ordering to the list of keys or values. The ordering depends on the internal implementation of the hash, which can depend both on the particular version of Perl you are using, the size of the hash, and a random factor. With that caveat in mind, the order of items in a hash is the same for `keys`, `values`, and `each`. Modifying the hash may change the order, but you can rely on that order if the hash remains the same.

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 hash.

Reset a hash's iterator with the use of `keys` or `values` in void context:

```
# reset hash iterator
keys %addresses;

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

You should also ensure that you do not call any function which may itself try to iterate over the hash with `each`.

The single hash iterator is a well-known caveat, but it doesn't come up as often as you might expect. Be cautious, but use `each` when you need it.

Hash Slices

As with arrays, you may access a list of elements of a hash in one operation. A *hash slice* is a list of keys or values of a hash. The simplest explanation is initialization of multiple elements of a hash used as an unordered set:

```
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 );
```

... except that the hash slice initialization does not *replace* the existing contents of the hash.

You may retrieve multiple values from a hash with a slice:

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

Note that, as with array slices, the sigil of the hash changes to indicate list context. You can still tell that `%addresses` is a hash by the use of the curly braces to indicate keyed access.

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.

The choice between the two approaches depends on your merge strategy. What if the same key occurs in both hashes? The hash slice approach always overwrites existing key/value pairs in `%addresses`.

The Empty Hash

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

```
use Test::More 'no_plan';

my %empty;
nok( %empty, 'empty hash should evaluate to 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'
);

...
```

In scalar context, a hash evaluates to a string which represents the number of hash buckets used out of the number of hash buckets allocated. This is rarely useful, as it represents internal details about hashes that are almost always meaningless to Perl programs. You can safely ignore it.

In list context, a hash evaluates to a list of key/value pairs similar to what you receive from the `each` operator. However, you *cannot* iterate over this list the same way you can iterate over the list produced by `each`, because there is no internal hash iteration provided when evaluating a hash in list contextThe loop will loop forever, unless the hash is empty..

Hash Idioms

Hashes have several uses, such as finding unique elements of lists or arrays. Because each key exists only once in a hash, assigning the same key to a hash multiple times stores only the most recent key:

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

The use of the `undef` operator with the hash slice sets the values of the hash to `undef`. This is the cheapest way to determine if an item exists in a set.

Hashes are also useful for counting elements, such as a list of IP addresses in a logfile:

```
my %ip_addresses;

while (my $line = <$logfile>)
{
    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, it creates a value (`undef`) and immediately increments it to one.

A variant of this strategy works very well for caching, where you might want to store the result of an expensive calculation with little overhead to store or fetch:

```
{
    my %user_cache;

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

This *orcish maneuver*Or-cache, if you like puns. returns the value from the hash, if it exists. Otherwise, it calculates the value, caches it, and then returns it. Beware that the boolean-or assignment operator (`||=`) operates on boolean values; if your cached value evaluates to false in a boolean context, use the defined-or assignment operator (`//=`) instead:

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

This lazy orcish maneuver tests for the definedness of the cached value, not its boolean truth. The

defined-or assignment operator is new in Perl 5.10.

Hashes can also collect named parameters passed to functions. If your function takes several arguments, you can use a slurpy hash (*parameter_slurping*) to gather key/value pairs into a single hash:

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

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

You can even set default parameters with this approach:

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

... or include them in the initial declaration and assignment itself:

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

... as subsequent declarations of the same key with a different value will overwrite the previous values.

Locking Hashes

One drawback of hashes is that their keys are barewords which offer little typo protection (especially compared to the function and variable name protection offered by the `strict` pragma). The core module `Hash::Util` provides mechanisms to restrict the modification of a hash or the keys allowed in the hash.

To prevent someone from accidentally adding a hash key you did not intend (presumably with a typo or with data from untrusted input), use the `lock_keys()` function to restrict the hash to its current set of keys. Any attempt to add a key/value pair to the hash where the key is not in the allowed set of keys will raise an exception.

Of course, anyone who needs to do so can always use the `unlock_keys()` function to remove the

protection, so do not rely on this as a security measure against misuse from other programmers.

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()`.

Coercion

Unlike other languages, where a variable can hold only a particular type of value (a string, a floating-point number, an object), Perl relies on the context of operators to determine how to interpret values (*value_contexts*). If you treat a number as a string, Perl will do its best to convert that number into a string (and vice versa). This process is *coercion*.

By design, Perl attempts to do what you mean (*DWIM* stands for *do what I mean*), though you must be specific about your intentions.

Boolean Coercion

Boolean coercion occurs when you test the *truthiness* of a valueTruthiness is like truthfulness if you squint and say "Yeah, that's true, but....", such as in a `if` or `while` condition. Numeric 0 is false. The undefined value is false. The empty string is false, and so is the string `'0'`. The strings `'0.0'` and `'0e'` are *true*, however.

All other values are true, including the idiomatic string `'0 but true'`. In the case of a scalar with both string and numeric portions (*dualvars*), Perl 5 prefers to check the string component for boolean truth. `'0 but true'` does evaluate to zero numerically, but is not the empty string, so it evaluates to true in boolean context.

String Coercion

String coercion occurs when using string operators such as comparisons (`eq` and `cmp`, for example), concatenation, `split`, `substr`, and regular expressions. It also occurs when using a value as a hash key. The undefined value stringifies to an empty string, but it produces a "use of uninitialized value" warning. Numbers *stringify* to strings containing their values. That is, the value `10` stringifies to the string `10`, such that you can `split` a number into individual digits:

```
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 as an array or list index. The undefined value *numifies* to zero, though it produces a "Use of uninitialized value" warning. Strings which do not begin with numeric portions also numify to zero, and they produce an "Argument isn't numeric" warning. Strings which begin with characters allowed in numeric literals numify to those values; that is, `10 leptons leaping` numifies to `10` the same way that `6.022e23 moles marauding` numifies to `6.022e23`.

The core module `Scalar::Util` contains a `looks_like_number()` function which uses the same parsing rules as the Perl 5 grammar to extract a number from a string.

Note that the strings `Inf` and `Infinity` represent the infinite value and behave as numbers, in the sense that numifying them does not produce the "Argument isn't numeric" warning.

Reference Coercion

In certain circumstances, treating a value as a reference turns that value *into* a reference. This process of autovivification (*autovivification*) can be useful for nested data structures. It occurs when you use a dereferencing operation on a non-reference:

```
my %users;

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

Although the hash never contained values for `Bradley` and `Jack`, Perl 5 helpfully created hash references for those values, then assigned them each a key/value pair keyed on `id`.

Cached Coercions

Perl 5's internal storage mechanism for values allows each value to have a stringification and a numificationThis is a simplification, but the gory details are truly gory.. Stringifying a numeric value does not replace the numeric value with a string. Instead, it *attaches* a stringified value to the value in addition to the numeric value. The same sort of operation happens when numifying a string value.

You almost never need to know that this happens--perhaps once or twice a decade, if anecdotal evidence is admissible.

Perl 5 may prefer one form over another. 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, but know that caching does occur and you may be able to diagnose an odd situation when it occurs.

Dualvars

The caching of string and numeric values allows for the use of a rare-but-useful feature known as a *dualvar*, or a value that has divergent numeric and string values. The core module `Scalar::Util` provides a function `dualvar()` which allows you to create a value which has specified and divergent numeric and string values:

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

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

Packages

A *namespace* in Perl is a mechanism which associates and encapsulates various named entities within a named category. It's like your family name or a brand name, except that it implies no relationship between entities other than categorization with that name. (Such a relationship often exists, but it does not have to exist.)

A *package* in Perl 5 is a collection of code in a single namespace. In a sense, a package and a namespace are equivalent; the package represents the source code and the namespace represents

the entity created when Perl parses that codeThis distinction may be subtle..

The `package` keyword 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. With this code as written, you can refer to the `@boxes` variable from the `main` namespace only by its *fully qualified* name, `@MyCode::boxes`. Similarly, you can call the `add_box()` function only by `MyCode::add_box()`. A fully qualified name includes its complete package name.

The default package is the `main` package. If you do not declare a package explicitly, whether in a one-liner on a command-line or in a standalone Perl program or even in a *.pm* file on disk, the current package will be the `main` package.

Besides a package name (`main` or `MyCode` or any other allowable identifier), a package has a version and three implicit methods, `VERSION()`, `import()`, and `unimport()`.

The package's version is a series of numbers contained in a package global named `$VERSION`. By convention, versions tend to be a series of integers separated by dots, as in `1.23` or `1.1.10`, where each segment is an integer, but there's little beyond convention.

Perl 5.12 introduced a new syntax intended to simplify version numbers. If you can write code that does not need to run on earlier versions of Perl 5, you can avoid a lot of unnecessary complexity:

```
package MyCode 1.2.1;
```

In 5.10 and earlier, the simplest way to declare the version of a package is:

```
package MyCode;

our $VERSION = 1.21;
```

The `VERSION()` method is available to every package; they inherit it from the `UNIVERSAL` base class. It returns the value of `$VERSION`. You may override it if you wish, though there are few reasons to do so. Obtaining the version number of a package is easiest through the use of the `VERSION()` method:

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

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


Every `package` declaration causes Perl to perform two tasks. It creates a new namespace if that

namespace does not already exist. It also tells the parser to put all subsequent package global symbols (global variables and functions) into 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 { ... }

package main;

Pack::first_sub();

package Pack;

sub second_sub { ... }

package main;

Pack::second_sub();
```

... or by fully qualifying function names at the point of declaration:

```
# implicit
package main;

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

Perl 5 packages are so open that you can add to them at any time during compilation or run time, or from separate files. Of course, that can be confusing, so avoid it when possible.

Namespaces can have as many levels as you like for organizational purposes. These are not hierarchical; there's no technical relationship between packages--only a semantic relationship to *readers* of the code.

It's common to create a top-level namespace for a business or a project. This makes a convenient organizational tool not only for reading code and discovering the relationships between components but also to organizing code and packages on disk. Thus:

* `StrangeMonkey` is the project name

* `StrangeMonkey::UI` contains the top-level user interface code

* `StrangeMonkey::Persistence` contains the top-level data management code

* `StrangeMonkey::Test` contains the top-level testing code for the project

... and so on.

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;
```

You probably expect that, outside of the function, `$name` contains `Chuck`, even though the value passed into the function gets reversed into `kcuhC`--and that's what happens. The `$name` outside the function is a separate scalar from the `$name` inside the function, and each one has a distinct copy of the string. Modifying one has no effect on the other.

This is useful and desirable default behavior. If you had to make explicit copies of every value before you did anything to them which could possibly cause changes, you'd write lots of extra, unnecessary code to defend against well-meaning but incorrect modifications.

Other times it's useful to modify a value in place sometimes as well. If you have a hash full of data that you want to pass to a function to update or to delete a key/value pair, creating and returning a new hash for each change could be troublesome (to say nothing of inefficient).

Perl 5 provides a mechanism by which you can refer to a value without making a copy of that value. Any changes made to that *reference* will update the value in place, such that *all* references to that value will see the new value. A reference is a first-class, built-in scalar data type in Perl 5. It's not a string, an array, or a hash. It's a scalar 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. Thus you can take a reference to `$name` from the previous example:

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

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

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

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

The double scalar sigil dereferences a scalar reference.

This example isn't useful in the obvious case; why not have the function return the modified value directly? Even so, scalar references are useful when processing *large* scalars; copying the contents of

those scalars can use a lot of time and memory.

Complex references may require a curly-brace block to disambiguate portions of the expression. This is optional for simple dereferences, though it can be messy:

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

If you forget to dereference a scalar reference, it will stringify or numify. The string value will be of the form `SCALAR(0x93339e8)`, and the numeric value will be the `0x93339e8` portion. This value encodes the type of reference (in this case, SCALAR) and the location in memory of the reference.

Perl does not offer native access to memory locations. The address of the reference is a value used as a mostly-unique identifier, as a reference does not necessarily have a name. Unlike pointers in a language such as C, you cannot modify the address or treat it as an address into memory.

These addresses are only mostly unique because Perl may reuse storage locations if its garbage collector has reclaimed an unreferenced reference.

Array References

You can also create references to arrays, or *array references*. This is useful for several reasons:

* To pass and return arrays from functions without flattening

* To create multi-dimensional data structures

* To avoid unnecessary array copying

* To hold anonymous data structures

To take a reference to a declared array, use the reference operator:

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

Now `$cards_ref` contains a reference to the array. Any modifications made through `$cards_ref` will modify `@cards` as well, and vice versa.

You may access the entire array as a whole with the @ sigil, whether to flatten the array into a list or count the number of elements it contains:

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

You may also access individual elements by using 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` from which you wish to access a single element.

An alternate syntax is available, where you prepend another scalar sigil to the array reference. It's

shorter, but somewhat less readable, to write `my $first_card = $$cards_ref[0];`.

You can slice an array through its reference with the curly-brace dereference grouping syntax:

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

In this case, you *may* omit the curly braces, but the visual grouping they (and the whitespace) provide only helps readability in this case.

You may also create anonymous arrays in place without using named arrays. Surround a list of values or expressions with square brackets:

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

This array reference behaves the same as named array references, except that anonymous array references *always* create a new reference, while taking a reference to a named array always refers to the *same* array with regard to scoping. That is to say:

```
my @meals      = qw( waffles 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( waffles 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 reference, the `@meals` array flattens in list context.

Hash References

To create a *hash reference*, use the reference operator on a named hash:

```
my %colors = (
    black  => 'negro',
    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>;
```

You may access individual values of the hash (to store, delete, check the existence of, or retrieve) by

using the dereferencing arrow:

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

As with array references, you may eschew the dereferencing arrow for a prepended scalar sigil: `$$colors_ref{$color}`, though the arrow is often much clearer.

You may also use hash slices by reference:

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

Note the use of curly brackets to denote a hash indexing operation and the use of the array sigil to denote a list operation on the reference.

You may create anonymous hashes in place with curly braces:

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

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

A common novice typo is to assign an anonymous hash to a standard hash. This 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 5 supports *first-class functions*. A function is a data type just as is an array or hash, at least when you use *function references*. This feature enables many advanced features (*closures*). As with other data types, you may create a function reference by using the reference operator on the name of a function:

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

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

Note that you must use the *function sigil* (`&`) with the name of the function. If you leave this off, you will take a reference to whatever the function itself returns.

You may also create anonymous functions:

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

The use of the `sub` keyword *without* a name compiles the function as normal, but does not install it in the current namespace. The only way to access this function is through the reference.

You may invoke the function reference with the dereferencing arrow:

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

Think of the empty parentheses as performing an invocation dereferencing operation in the same way that square brackets perform an indexed lookup and curly brackets perform a hash lookup. You may 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 most useful when you've already performed a method lookup:

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

You may see an alternate invocation syntax for function references which uses the function sigil (`&`) instead of the dereferencing arrow. Avoid this syntax; it has implications for implicit argument passing.

Filehandle References

Filehandles can be references as well. When you use `open`'s (and `opendir`'s) lexical filehandle form, you deal with filehandle references. Stringifying this filehandle produces something of the form `GLOB(0x8bda880)`.

Internally, these filehandles are objects of the class `IO::Handle`. When you load that module, you can call methods on filehandles:

```
use IO::Handle;
use autodie;

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

You may see old code which takes references to typeglobs, such as:

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

This idiom predates lexical filehandles, introduced as part of Perl 5.6.0 in March 2000... so you know how old that code is.. 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 represent global data anyhow. For all other filehandles, prefer lexical filehandles.

Besides the benefit of using lexical scope instead of package or global scope, lexical filehandles allow you to manage the lifespan of filehandles. This is a nice feature of how Perl 5 manages memory and scopes.

Reference Counts

How does Perl know when it can safely release the memory for a variable and when it needs to keep it around? How does Perl know when it's safe to close the file opened in this inner scope:

```
use autodie;
use IO::Handle;

sub show_off_scope
{
    say 'file not open';

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

    say 'file closed here';
}
```

Perl 5 uses a memory management technique known as *reference counting*. Every value in the program has an attached counter. 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.

Within the inner block in the example, there's one `$fh`. (Multiple lines in the source code refer to it, but there's only one *reference* to it; `$fh` itself.) `$fh` is only in scope in the block and does not get assigned to anything outside of the block, so when the block ends, its reference count reaches zero. The recycling of `$fh` calls an implicit `close()` method on the filehandle, and so the file gets closed.

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. (Though see *circular_references* for one caveat.)

Nested Data Structures

Perl's aggregate data types--arrays and hashes--allow you to store scalars indexed by integers or string keys. Perl 5's references (*references*) allow you to access aggregate data types indirectly, through special scalars. Nested data structures in Perl, such as an array of arrays or a hash of hashes, are possible through the use of references.

Declaring Nested Data Structures

A simple declaration of an array of arrays might be:

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

... and a simple declaration of a hash of hashes might be:

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

Perl allows but does not require the trailing comma so to ease adding new elements to the list.

Accessing Nested Data Structures

Accessing elements in nested data structures uses Perl's reference syntax. The sigil denotes the amount of data to retrieve, and 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 $breaky_side = $meals{breakfast}->{side};
```

In the case of a nested data structure, the only way to nest a data structure is through references, thus the arrow is superfluous. This code is equivalent and clearer:

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

You can avoid the arrow in every case except invoking a function reference stored in a nested data structure, where the arrow invocation syntax is the clearest mechanism of invocation.

Accessing components of nested data structures as if they were first-class arrays or hashes requires disambiguation blocks:

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

Similarly, slicing a nested data structure requires additional punctuation:

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

The use of whitespace helps, but it does not entirely eliminate the noise of this construct. Sometimes using temporary variables can clarify:

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

`perldoc perldsc`, the data structures cookbook, gives copious examples of various types of data structures available in Perl with their syntax.

Autovivification

Perl's expressivity extends to nested data structures. When you attempt to write to a component of a nested data structure, Perl will create the path through the data structure to that piece if it does not exist:

```
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, treating an undefined value as if it were a hash reference in a nested data structure will create intermediary hashes, keyed appropriately:

```
my %hohoh;

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

This behavior is *autovivification*, and it's more often useful than it isn't. Its benefit is in reducing the initialization code of nested data structures. Its drawback is in its inability to distinguish between the honest intent to create missing elements in nested data structures and typos.

The `autovivification` pragma on the CPAN (*pragmas*) lets you disable autovivification in a lexical scope for specific types of operations; it's worth your time to consider this in large projects, or projects with multiple developers.

You can also check for the existence of specific hash keys and the number of elements in arrays before dereferencing each level of a complex data structure, but that can produce tedious, lengthy code which many programmers prefer to avoid.

You may wonder at the contradiction between taking advantage of autovivification while enabling `strictures`. The question is one of balance. is it more convenient to catch errors which change the behavior of your program at the expense of disabling those error checks for a few well-encapsulated symbolic references? Is it more convenient to allow data structures to grow rather than specifying their size and allowed keys?

The answer to the latter question depends on your specific project.

Debugging Nested Data Structures

The complexity of Perl 5's dereferencing syntax combined with the potential for confusion with multiple levels of references can make debugging nested data structures difficult. Two good options exist for visualizing them.

The core module `Data::Dumper` can stringify values of arbitrary complexity into Perl 5 code:

```
use Data::Dumper;

print Dumper( $my_complex_structure );
```

This is useful for identifying what a data structure contains, what you should access, and what you accessed instead. `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 5 code, it also produces verbose output. Some developers prefer the use of the `YAML::XS` or `JSON` modules for debugging. You have to learn a different format to understand their outputs, but their outputs can be much clearer to read and to understand.

Circular References

Perl 5's memory management system of reference counting (*reference_counts*) has one drawback apparent to user code. Two references which end up pointing to each other form a *circular reference* that Perl cannot destroy on its own. Consider a biological model, where each entity has two parents and can have children:

```
my $alice  = { mother => '',     father => '',      children => [] };
my $robert = { mother => '',     father => '',      children => [] };
my $cianne = { mother => $alice, father => $robert, children => [] };

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

Because both `$alice` and `$robert` contain an array reference which contains `$cianne`, and because `$cianne` is a hash reference which contains `$alice` and `$robert`, Perl can 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.

You must either break the reference count manually yourself (by clearing the children of `$alice` and `$bob` or the parents of `$cianne`), or take advantage of a feature called *weak references*. A weak reference is a reference which does not increase the reference count of its referent. Weak references are available through the core module `Scalar::Util`. Export the `weaken()` function and use it on a reference to prevent the reference count from increasing:

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

my $alice  = { mother => '',     father => '',      children => [] };
my $robert = { mother => '',     father => '',      children => [] };
my $cianne = { mother => $alice, father => $robert, children => [] };

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

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

With this accomplished, `$cianne` will retain references to `$alice` and `$robert`, but those references will not by themselves prevent Perl's garbage collector from destroying those data structures. You rarely have to use weak references if you design your data structures correctly, but they're useful in a few situations.

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 as well as the relationship of various pieces, not to mention the syntax required to access various portions, can be high. Beyond two or three levels of nesting, consider whether modeling various components of your system with classes and objects (*moose*) will allow you to operate on your data within encapsulation boundaries.

Sometimes bundling data with behaviors appropriate to that data can clarify code.