Like a spoken language, the whole of Perl is a combination of several smaller but interrelated parts. 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. These names all begin with a letter or an underscore and may optionally include any combination of letters, numbers, and underscores. When the `utf8` pragma (*unicode*) is in effect, you may use any UTF-8 word characters in identifiers. These are all valid Perl identifiers:

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

sub anAwkwardName3;

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

These are invalid Perl identifiers:

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

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

*Names exist primarily for the benefit of the programmer*. These rules apply only to literal names which appear as-is in your source code, such as `sub fetch_pie` or `my $waffleiron`. Only Perl's parser enforces the rules about identifier names.

Perl's dynamic nature allows you to refer to entities with names generated at runtime or provided as input to a program. These *symbolic lookups* provide flexibility at the expense of some safety. In particular, invoking functions or methods indirectly or looking up symbols in a namespace lets you bypass Perl's parser.

Doing so can produce confusing code. As Mark Jason Dominus recommends so effectively http://perl.plover.com/varvarname.html, use a hash (*hashes*) or nested data structure ( *nested_data_structures*).

### Variable Names and Sigils

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

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

These sigils provide a visual namespacing for variable names. It's possible--though confusing--to declare multiple variables of the same name with different types:

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

Though Perl won't get confused, people reading this code will.

Perl 5's sigils are *variant sigils*. As context determines how many items you expect from an operation or what type of data you expect to get, so the sigil governs how you manipulate the data of a variable. For example, to access a single element of an array or a hash, you must use the scalar sigil ($):

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

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

The parallel with amount context is important. Using a scalar element of an aggregate as an *lvalue* (the target of an assignment, on the 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 (@) and imposes list context... even if the list itself has zero or one elements:

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

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

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.

## Namespaces

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

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

While 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 core pragmas (*pragmas* ), such as `strict` and `warnings`. This is a policy enforced primarily by community guidelines.

All namespaces in Perl 5 are globally visible. When Perl looks up a symbol in `DessertShop::IceCream::Freezer`, it looks in the `main::` symbol table for a symbol

representing the `DessertShop::` namespace, then in there for the `IceCream::` namespace, and so on. The `Freezer::` is visible from outside of the `IceCream::` namespace. The nesting of the former within the latter is only a storage mechanism, and implies nothing further about relationships between parent and child or sibling packages. Only a programmer can make *logical* relationships between entities obvious--by choosing good names and organizing them well.

## Variables

A *variable* in Perl is a storage location for a value (*values*). While a trivial program can manipulate values directly, most programs work with variables to simplify the logic of the code. A variable represents values; it's easier to explain the Pythagorean theorem in terms of the variables a, b, and c than by intuiting its principle by producing a long list of valid values. This concept may seem basic, but effective programming requires you to manage the art of balancing the generic and reusable with the specific.

### Variable Scopes

Variables are visible to portions of your program depending on their scope (*scope*). Most of the variables you will encounter have lexical scope (*lexical_scope*). *Files* themselves provide their own lexical scopes, such that the `package` declaration on its own does not create a new scope:

```
package Store::Toy;

my $discount = 0.10;

package Store::Music;

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

As of Perl 5.14, you may provide a block to the `package` declaration. This syntax *does* provide a lexical scope:

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

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

package Store::BoardGame;

# $discount still not available
```

### Variable Sigils

The sigil of the variable in a declaration determines the type of the variable: scalar, array, or hash. The sigil used when accessing a variable varies depending on what you do to the variable. For example, you declare an array as `@values`. Access the first element--a single value--of the array with `$values[0]`. Access a list of values from the array with `@values[ @indices ]`.

### Anonymous Variables

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

### Variables, Types, and Coercion

A variable in Perl 5 represents both a value (a dollar cost, available pizza toppings, guitar shops with phone numbers) and the container which stores that value. Perl's type system deals with *value types* and *container types*. While a variable's *container type*--scalar, array, or hash--cannot change, Perl is flexible about a variable's value type. You may store a string in a variable in one line, append to that variable a number on the next, and reassign a reference to a function (*function_references*) on the third.

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

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

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

This relationship between variable types, sigils, and context is essential.

## Values

The structure of a program depends heavily on the means by which you model your data with appropriate variables.

Where variables allow the abstract manipulation of data, the values they hold make programs concrete and useful. The more accurate your values, the better your programs. These values are data--your aunt's name and address, the distance between your office and a golf course on the moon, or the weight of all of the cookies you've eaten in the past year. Within your program, the rules regarding the format of that data are often strict. Effective programs need effective (simple, fast, most compact, most efficient) ways of representing their data.

### Strings

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

To represent a literal string in your program, surround it with a pair of quoting characters. The most common *string delimiters* are single and double quotes:

```
my $name    = B<'Donner Odinson, Bringer of Despair'>;
```

```
my $address = B<"Room 539, Bilskirnir, Valhalla">;
```

Characters in a *single-quoted string* represent themselves literally, with two exceptions. Embed a single quote inside a single-quoted string by escaping the quote with a leading backslash:

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

You must also escape any backslash at the end of the string to avoid escaping the closing delimiter and producing a syntax error:

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

Any other backslash will be part of the string as it appears, unless two backslashes are adjacent, in which case the first will escape the second:

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

A *double-quoted string* has several more special characters available. For example, you may encode otherwise invisible whitespace characters in the string:

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

This demonstrates a useful principle: the syntax used to declare a string may vary. You can represent a tab within a string with the `\t` escape or by typing a tab directly. Within Perl's purview, both strings behave the same way, even though the specific representation of the string may differ in the source code.

A string declaration may cross logical newlines; these two declarations are equivalent:

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

These sequences are often easier to read than their whitespace equivalents.

Perl strings have variable lengths. As you manipulate and modify strings, Perl will change their sizes as appropriate. For example, you can combine multiple strings into a larger string with the *concatenation* operator `.`:

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

This is effectively the same as if you'd initialized the string all at once.

You may also *interpolate* the value of a scalar variable or the values of an array within a

double-quoted string, such that the *current* contents of the variable become part of the string as if you'd concatenated them:

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

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

Include a literal double-quote inside a double-quoted string by *escaping* it (that is, preceding it with a leading backslash):

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

When repeated backslashing becomes unwieldy, use an alternate *quoting operator* by which you can choose an alternate string delimiter. The q operator indicates single quoting, while the qq operator provides double quoting behavior. The character immediately following the operator determines the characters used to delimit the strings. If the character is the opening character of a balanced pair--such as opening and closing braces--the closing character will be the final delimiter. Otherwise, the character itself will be both the starting and ending delimiter.

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

When declaring a complex string with a series of embedded escapes is tedious, use the *heredoc* syntax to assign one or more lines of a string:

```
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.
We instead, born out of time, remain perfect and
perfectly self-aware. We only suffer change as we
pursue it. It is against our nature. We rebel
against that change. Shall we consider them
greater for it?"
END_BLURB
```

The `<<'END_BLURB'` syntax has three parts. The double angle-brackets introduce the heredoc. The quotes determine whether the heredoc obeys single- or double-quoted behavior. The default behavior is double-quoted interpolation. `END_BLURB` 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* start at the beginning of the line:

```
sub some_function {
    my $ingredients =<<'END_INGREDIENTS';
    Two eggs
    One cup flour
    Two ounces butter
    One-quarter teaspoon salt
```

```
    One cup milk
    One drop vanilla
    Season to taste
END_INGREDIENTS
}
```

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

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

## Unicode and Strings

*Unicode* is a system for representing the characters of 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.

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

Sequences of Unicode characters

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

Sequences of octets

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

Why *octet* and not *byte*? Assuming that one character fits in one byte will cause you no end of Unicode grief. Separate the idea of memory storage from character representation.

Unicode strings and binary strings look similar. Each has a `length()`. Each supports standard string operations 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. You must add a specific meaning to that data.

### Character Encodings

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

To avoid most Unicode problems, always decode to and from the appropriate encoding at the inputs and outputs of your program.

Perl 5.12 supports the Unicode 5.2 standard, while Perl 5.14 supports Unicode 6.0. If you need to care about the differences between Unicode versions, you probably already know to see http://unicode.org/versions/.

### Unicode in Your Filehandles

When you tell Perl that a specific filehandle (*files*) works with encoded text, Perl will convert the incoming octets to Unicode strings automatically. To do this, add an IO layer to the mode of the `open` builtin. An *IO layer* wraps around input or output and converts the data. In this case, the `:utf8` layer decodes UTF-8 data:

```
use autodie;

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

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

You may also modify an existing filehandle with `binmode`, whether for input or output:

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

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

```
my $from_utf8 = decode('utf8', $data);
my $to_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 and identifiers:

```
use utf8;

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

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

To *write* this code, your text editor must understand UTF-8 and you must save the file with the appropriate encoding.

Within double-quoted strings, you may 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}";
```

Some Unicode characters have names, and these names are often clearer to read than Unicode numbers. Use the `charnames` pragma to enable them and 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 will contain Unicode characters. When you print Unicode characters, Perl will encode the string using UTF-8, because Latin-1 cannot represent the entire set of Unicode characters--Latin-1 is a subset of UTF-8.

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 character.

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

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

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

The string literal has several possible scenarios:

* It is an ASCII string literal and contains octets.

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

* It is a Latin-1 string literal with no explicit encoding and contains octets.

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

The string literal contains octets.

* It is a non-ASCII string literal with the `utf8` or `encoding` pragma in effect and contains Unicode characters.

```
use utf8;
my $hello = "KuirabE<aacute>, ";
```

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, for example a concatenation appending UTF-8 data to Latin-1 data, then the resulting sequence of octets makes sense in *neither* encoding. This could happen if the user entered a name as UTF-8 data and the greeting were a Latin-1 string literal, but the program decoded neither.

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

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

Perl 5.12 added a feature, `unicode_strings`, which enables Unicode semantics for all string operations within its scope. Perl 5.14 improved this feature; if you work with Unicode in Perl, it's worth upgrading to at least Perl 5.14.

## Numbers

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

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

The emboldened characters are the numeric prefixes for binary, octal, and hex notation respectively. Be aware that a leading zero on an integer *always* indicates octal mode.

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

You may not use commas to separate thousands in numeric literals, lest the parser interpret the commas as comma operators. Instead, use underscores within the number. 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.

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. In the rare circumstances where you need to know if something looks like a number to Perl, use the `looks_like_number` function from the core module `Scalar::Util`. This function returns a true value if Perl will consider the given argument numeric.

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

## Undef

Perl 5's `undef` value represents an unassigned, undefined, and unknown value. Declared but undefined scalar variables contain `undef`:

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

`undef` evaluates to false in boolean context. Evaluating `undef` in a string context--such as interpolating it into a string--produces an `uninitialized value` warning:

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

... produces:

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

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

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

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

## The Empty List

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

If you find that concept confusing right now, fear not. As you understand how Perl's fundamental design features fit together in practice, it will make more sense.

## 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();
```

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

Use the range operator to create lists of literals in a compact form:

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

Use the `qw()` operator to split a literal string on whitespace to produce a list of strings:

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

Perl will emit a warning if a `qw()` contains a comma or the comment character (#), because not only are such characters rare in a `qw()`, their presence 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. Arrays are containers. You may store a list in an array and you may coerce an array to a list, but they are separate entities. For example, indexing into a list always occurs in list context. Indexing into an array can occur in scalar context (for a single element) or list context (for a slice):

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

    say defined $context
         ? $context
             ? 'list'
```

```
            : 'scalar'
        : 'void';
    return 0;
}

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

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

## Control Flow

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

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

Perl's *control flow directives* change the order of execution--what happens next in the program--depending on the values of their expressions.

### Branching Directives

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

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

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

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

While the block form requires parentheses around its condition, the postfix form does not.

The conditional expression may consist of multiple subexpressions, as long as it evaluates to a single top-level expression:

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

In the postfix form, adding parentheses can clarify the intent of the code at the expense of visual cleanliness:

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

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

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

Like `if`, `unless` also has a block form, though many programmers avoid it, as it rapidly becomes difficult to read with complex conditionals:

```
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 (@_) { ... }
}
```

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:

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

However, the implied double negative of using `unless` with an `else` block can be confusing. This

example may be the only place you ever see it.

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

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

... though the double negative implied by the presence of the `else` block suggests inverting the conditional.

One or more `elsif` directives may follow an `if` block form and may precede any single `else`:

```
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();
}
```

An `unless` chain may also use an `elsif` blockGood luck deciphering that!. There is no `elseunless`.

Writing `else if` is a syntax errorLarry prefers `elsif` for aesthetic reasons, as well the prior art of the Ada programming language.:

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

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

### The Ternary Conditional Operator

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

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

The conditional expression precedes the question mark character (?) and the colon character (:) separates the alternatives. The alternatives are expressions of arbitrary complexity--including other ternary conditional expressions.

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.

**Short Circuiting**

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

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

done_testing();
```

The return value of `ok()` (*testing*) is the boolean value obtained by evaluating the first argument, so this code prints:

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

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

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

This example prints:

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

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

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

This example prints:

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

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, as in the case where using an undefined value might raise a warning:

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

## Context for Conditional Directives

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

Perl 5 has no single true value, nor a single false value. Any number which evaluates to 0 is false. This includes `0`, `0.0`, `0e0`, `0x0`, and so on. The empty string (`''`) and `'0'` evaluate to a false value, but the strings `'0.0'`, `'0e0'`, and so on do not. The idiom `'0 but true'` evaluates to 0 in numeric context but true in boolean context, thanks to its string contents.

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

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

## Looping Directives

Perl provides several directives for looping and iteration. The *foreach*-style loop evaluates an expression which produces a list and executes a statement or block until it has 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 `$_` (*default_scalar_variable*) to each in turn. Perl executes the block for each integer and prints the squares of the integers.

Many Perl programmers refer to iteration as `foreach` loops, but Perl treats the names `foreach` and `for` interchangeably. The subsequent code determines the type and behavior of the loop.

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

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

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

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

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

```
my $i = 'cow';

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

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

This localization occurs even if you do not redeclare the iteration variable as a lexical... but *do* declare your iteration variables as lexicals to reduce their scope.:

```
my $i = 'horse';

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

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

### Iteration and Aliasing

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

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

$_ **= 2 for @nums;

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

...

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

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

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

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

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

You cannot use aliasing to modify *constant* values, however:

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

Instead Perl will produce an exception about modification of read-only values.

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

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

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

## Iteration and Scoping

Iterator scoping with the topic variable provides one common source of confusion. Consider a function `topic_mangler()` which modifies `$_` on purpose. If code iterating over a list called `topic_mangler()` without protecting `$_`, debugging fun would ensue:

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

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

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

```
sub topic_mangler
{
    # 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* requires you to manage the conditions of iteration:

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

You must explicitly assign to an iteration variable in the looping construct, as this loop performs neither aliasing nor assignment to the topic variable. While any variable declared in the loop construct is scoped to the lexical block of the loop, there is no lexicalization of a variable declared outside of the loop construct:

```
my $i = 'pig';

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

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

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

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

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

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

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

All three subexpressions are optional. An infinite `for` loop might be:

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

## While and Until

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

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

Unlike the iteration `foreach`-style loop, the `while` loop's condition has no side effects by itself. That is, 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, use a *destructive update* of the `@values` array by modifying the array with each loop iteration:

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

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

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

This loop will exit as soon as it reaches an element that evaluates to a false value, not necessarily when it has exhausted the array. That may be the desired behavior, but is often surprising to novices.

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

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

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

```
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 a false value in a scalar context--a blank line or a line which contained only the character `0`--would end the loop. The `readline` (<>) operator returns an undefined value only when it has reached the end of the file.

Use the `chomp` builtin to remove line-ending characters from each line. Many novices forget this.

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

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

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

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

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

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

A `do` block parses as a single expression which may contain several expressions. Unlike the `while` loop's block form, the `do` block with a postfix `while` or `until` will execute its body at least once. This construct is less common than the other loop forms, but no less powerful.

## Loops within Loops

You may nest loops within other loops:

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

When you do so, declare named iteration variables! The potential for confusion with the topic variable and its scope is too great otherwise.

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, 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 starts with the comment character #, write:

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

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

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

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

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

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

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

    ...
}
```

Using loop control statements in nested loops can be confusing. If you cannot avoid nested loops--by extracting inner loops into named functions--use a *loop label* to clarify:

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

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

## Continue

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

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

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

## 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 expression in scalar context, and always assigns to the topic variable:

```
given (my $username = find_user())
{
    is( $username, $_, 'topic auto-assignment' );
}
```

`given` also lexicalizes the topic variable:

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

sub mouse_to_man { s/mouse/man/ }
```

`given` is most useful when combined with `when` (*smart_match*). `given` *topicalizes* a value within a block so that multiple `when` statements can match the topic against expressions using *smart-match* semantics. To write the Rock, Paper, Scissors game:

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

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

sub rock
{
    print "I chose rock.  ";

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

```
        default           { say $confused  };
    }
}

sub paper
{
    print "I chose paper.  ";

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

sub scissors
{
    print "I chose scissors.  ";

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

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

The CPAN module `MooseX::MultiMethods` provides another technique to simplify this code.

## Tailcalls

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

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

    return greet_person( $name );
}
```

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

Heavily recursive code (*recursion*), especially mutually recursive code, can consume a lot of memory. Tailcalls reduce the memory needed for internal bookkeeping of control flow and can make expensive algorithms tractable. Unfortunately, Perl 5 does not automatically perform this optimization; you have

to do it yourself when it's necessary.

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

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

    B<goto &greet_person>;
}
```

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

This technique is relatively rare; it's most useful when you want to hijack control flow to get out of the way of other functions inspecting `caller` (such as when you're implementing special logging or some sort of debugging feature), or when using an algorithm which requires a lot of recursion.

## Scalars

Perl 5's fundamental data type is the *scalar*, a single, discrete value. That value may be a string, an integer, a floating point value, a filehandle, or a reference--but it is always a single value. Scalars may be lexical, package, or global (*globals*) variables. You may only declare lexical or package variables. The names of scalar variables must conform to standard variable naming guidelines (*names*). Scalar variables always use the leading dollar-sign ($) sigil (*sigils*).

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

### Scalars and Types

A scalar variable can contain any type of scalar value without special conversions or casts, and the type of value stored in a variable can change:

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

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

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

```
my $zip_code       = 97006;
my $city_state_zip = 'Beaverton, Oregon' . ' ' . $zip_code;
```

You may also use mathematical operations on strings:

```
my $call_sign = 'KBMIU';
```

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

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

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

This magical string increment behavior has no corresponding magical decrement behavior. You can't 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. While `ZZ9` becomes `AAA0`, `ZZ09` becomes `ZZ10`--numbers wrap around while there are more significant places to increment, as on a vehicle odometer.

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

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

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

To allow coercion without data loss, Perl 5 scalars can contain both numeric and string components. 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 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.

Scalars do not contain 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 a distinction between 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 *first-class* data structures--the language supports them as a built-in data type--which store zero or more scalars. You can access individual members of the array by integer indexes, and you can add or remove elements at will. 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. `$cats[0]` is an unambiguous use of the `@cats` array, because postfix (*fixity*) square brackets (`[]`) always mean indexed access to an array.

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 in the array:

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

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

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

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

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

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

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

When the index matters less than the position of an element, use negative array indices instead. The last element of an array is available at the index `-1`. The second to last element of the array is available at index `-2`, and so on:

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

$# has another use: resize an array in place by *assigning* to it. Remember that Perl 5 arrays are mutable. They expand or contract as necessary. When you shrink an array, Perl will discard values which do not fit in the resized array. When you expand an array, Perl will fill the expanded positions with `undef`.

### Array Assignment

Assign to individual positions in an array directly by index:

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

If you assign to an index beyond the array's current bound, Perl will extend the array to account for the new size and will fill in all intermediary positions with `undef`. After the first assignment, the array will contain `undef` at positions 0, 1, and 2 and `Jack` at position 3.

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

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

... but remember that these parentheses *do not* create a list. Without parentheses, this would assign `Daisy` as the first and only element of the array, due to operator precedence (*precedence*).

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 imposes scalar context, while assigning to the array as a whole imposes list context.

To clear an array, assign an empty list:

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

`my @items = ();` is a longer and noisier version of `my @items` because freshly-declared arrays start out empty.

### Array Operations

Sometimes an array is more convenient as an ordered, mutable collection of items than as a mapping of indices to values. Perl 5 provides several operations to manipulate array elements without using indices.

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

```
my @meals;

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

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

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

Because `push` operates on a list, you can easily append the elements of one or more arrays to another with:

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

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

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

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

`unshift` prepends a list of elements to the start of the array and returns the new number of elements in the array. `shift` removes and returns the first element of the array.

Few programs use the return values of `push` and `unshift`.

The `splice` operator 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`. One effective use is removal of two elements from an array:

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

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

Prior to Perl 5.12, iterating over an array by index required a C-style loop. As of Perl 5.12, `each` can iterate over an array by index and value:

```
while (my ($index, $value) = each @bookshelf)
{
    say "#$index: $value";
    ...
}
```

## Array Slices

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

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

Array slices are useful for assignment:

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

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

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

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

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

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

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

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

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

Within the function, `@_` will contain seven elements, not two, because list assignment to arrays is *greedy*. An array will consume as many elements from the list as possible. After the assignment, `@cats` will contain *every* argument passed to the function. `@dogs` will be empty.

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 @numbers = (  1 .. 10,
              ( 11 .. 20,
              ( 21 .. 30 ) ) );
```

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

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

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

### Array Interpolation

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

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

Localize `$"` with a delimiter to ease your debuggingCredit goes to Mark Jason Dominus for this technique.:

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

## Hashes

A *hash* is a first-class Perl data structure which associates string keys with scalar values. In the same way that the name of a variable corresponds to a storage location, a key in a hash refers to a value. Think of a hash like you would a telephone book: use the names of your friends to look up their numbers. Other languages call hashes *tables*, *associative arrays*, *dictionaries*, or *maps*.

Hashes have two important properties: they store one scalar per unique key and they provide no specific ordering of keys.

### Declaring Hashes

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

```
my %favorite_flavors;
```

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

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

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

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

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

If you assign an odd number of elements to the hash, you will receive a warning to that effect. Idiomatic Perl often uses the *fat comma* operator (=>) to associate values with keys, as it makes the pairing more visible:

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

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

```
sub name { 'Leonardo' }

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

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

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

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

Assign an empty list to empty a hashYou may occasionally see `undef %hash`.:

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

## Hash Indexing

Access individual hash values with an indexing operation. Use a key (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<()>};
```

Novices often always quote string literal hash keys, but experienced developers elide the quotes whenever possible. In this way, the presence of quotes in hash keys signifies an intention to do something different.

Even Perl 5 builtins get the autoquoting treatment:

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

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

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

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

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

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

Hash keys can only be strings. Anything that evaluates to a string is an acceptable hash key. Perl will go so far as to coerce (*coercion*) any non-string into a string, such that 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`):

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

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

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

```
$addresses{Leibniz} = undef;

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

## Accessing Hash Keys and Values

Hashes are aggregate variables, but their pairwise nature offers many more possibilities for iteration: over the keys of a hash, the values of a hash, or pairs of keys and values. The `keys` operator produces a list of hash keys:

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

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

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

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

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

Unlike arrays, there is no obvious ordering to these lists. The ordering depends on the internal implementation of the hash, the particular version of Perl you are using, the size of the hash, and a random factor. Even so, the order of hash items is consistent between `keys`, `values`, and `each`. Modifying the hash may change the order, but you can rely on that order if the hash remains the

same.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. During such iteration, beware not to call any function which may itself try to iterate over the hash with `each`.

In practice this occurs rarely, but reset a hash's iterator with `keys` or `values` in void context when you need it:

```
# reset hash iterator
keys %addresses;

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

## Hash Slices

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

```
# %cats already contains elements
@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.

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

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

Hash slices make it easy to merge two hashes:

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

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

This is equivalent to looping over the contents of `%canada_addresses` manually, but is much shorter.

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

## The Empty Hash

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

```
use Test::More;

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

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

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

done_testing();
```

In scalar context, a hash evaluates to a string which represents the ratio of full buckets in the hash--internal details about the hash implementation that you can safely ignore.

In 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`, lest the loop will never terminate:

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

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

## Hash Idioms

Because each key exists only once in a hash, assigning the same key to a hash multiple times stores only the most recent key. Use this to find unique list elements:

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

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

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

```
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, Perl creates a value (`undef`) and immediately increments it to one, as the numification of `undef` produces the value 0.

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

```
{
    my %user_cache;

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

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

Perl 5.10 introduced the defined-or and defined-or assignment operators. Prior to 5.10, most code used the boolean-or assignment operator (`||=`) for this purpose. Unfortunately, some valid values evaluate to a false value in boolean context, so evaluating the *definedness* of values is almost always more accurate. This lazy orcish maneuver tests for the definedness of the cached value, not truthiness.

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

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

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

This approach allows you to set default values:

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

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

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

### Locking Hashes

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

To prevent someone from accidentally adding a hash key you did not intend (whether as a typo or from untrusted user input), use the `lock_keys()` function to restrict the hash to its current set of keys. Any attempt to add a new key to the hash will raise an exception. This is lax security suitable only for preventing accidents; anyone can use the `unlock_keys()` function to remove this protection.

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

A Perl variable can hold at various times values of different types--strings, integers, rational numbers, and more. Rather than attaching type information to variables, Perl relies on the context provided by operators (*value_contexts*) to know what to do with values. By design, Perl attempts to do what you meanCalled *DWIM* for *do what I mean* or *dwimmery*., though you must be specific about your intentions. If you treat a variable which happens to contain a number as a string, Perl will do its best to *coerce* that number into a string.

### Boolean Coercion

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

When a scalar has *both* string and numeric components (*dualvars*), Perl 5 prefers to check the string component for boolean truth. `'0 but true'` evaluates to zero numerically, but it is not an empty string, thus it evaluates to a true value in boolean context.

### String Coercion

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

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

### Numeric Coercion

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

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

The strings `Inf` and `Infinity` represent the infinite value and behave as numbers. The string `NaN` represents the concept "not a number". Numifying them produces no "Argument isn't numeric" warning.

### Reference Coercion

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

```
my %users;

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

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

### Cached Coercions

Perl 5's internal representation of values stores both string and numeric values. Stringifying a numeric value does not *replace* the numeric value. Instead, it *attaches* a stringified value, so that the representation contains *both* components. Similarly, numifying a string value populates the numeric component while leaving the string component untouched.

Certain Perl operations prefer to use one component of a value over another--boolean checks prefer strings, for example. If a value has a cached representation in a form you do not expect, relying on an implicit conversion may produce surprising results. You almost never need to be explicit about what you expectYour author can recall doing so twice in over a decade of programming Perl 5, but knowing that this caching occurs may someday help you diagnose an odd situation.

### Dualvars

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

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

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

## Packages

A Perl *namespace* associates and encapsulates various named entities within a named category, like your family name or a brand name. Unlike a real-world name, a namespace implies no direct relationship between entities. Such relationships may exist, but do not have to.

A *package* in Perl 5 is a collection of code in a single namespace. The distinction is subtle: the package represents the source code and the namespace represents the entity created when Perl parses that code.

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

```
package MyCode;

our @boxes;

sub add_box { ... }
```

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

The scope of a package continues until the next `package` declaration or the end of the file, whichever comes first. Perl 5.14 enhanced `package` so that you may provide a block which explicitly delineates the scope of the declaration:

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

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

Besides a name, a package has a version and three implicit methods, `import()` (*importing*), `unimport()`, and `VERSION()`. `VERSION()` returns the package's version number. This number is a series of numbers contained in a package global named `$VERSION`. By rough 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.

Perl 5.12 introduced a new syntax intended to simplify version numbers, as documented in `perldoc version::Internals`. These stricter version numbers must have a leading `v` character and at least three integer components separated by periods:

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

With Perl 5.14, the optional block form of a `package` declaration is:

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

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

```
package MyCode;

our $VERSION = 1.21;
```

Every package inherits a `VERSION()` method from the `UNIVERSAL` base class. You may override `VERSION()`, though there are few reasons to do so. This method returns the value of `$VERSION`:

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

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

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

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

## Packages and Namespaces

Every `package` declaration creates a new namespace if necessary and causes 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 { ... }
}
```

```
Pack::first_sub();

package Pack
{
    sub second_sub { ... }
}

Pack::second_sub();
```

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

```
# implicit
package main;

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

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

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

* `StrangeMonkey` is the project name

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

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

* `StrangeMonkey::Test` contains 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;
```

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

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

Yet sometimes it's useful to modify values in place. If you want to pass a hash full of data to a function to modify it, creating and returning a new hash for each change could be troublesome (to say nothing

of inefficient).

Perl 5 provides a mechanism by which to refer to a value without making a copy. Any changes made to that *reference* will update the value in place, such that *all* references to that value can reach the new value. A reference is a first-class scalar data type in Perl 5 which refers to another first-class data type.

## Scalar References

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

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

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

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

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

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

While in `@_`, parameters behave as *aliases* to caller variablesRemember that `for` loops produce a similar aliasing behavior., so you can modify them in place:

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

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

You usually don't want to modify values this way--callers rarely expect it, for example. Assigning parameters to lexicals within your functions removes this aliasing behavior.

Modifying a value in place, or returning a reference to a scalar can save memory. Because Perl copies values on assignment, you could end up with multiple copies of a large string. Passing around references means that Perl will only copy the references--a far cheaper operation.

Complex references may require a curly-brace block to disambiguate portions of the expression. You

may always use this syntax, though sometimes it clarifies and other times it obscures:

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

If you forget to dereference a scalar reference, Perl will likely coerce the reference. 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 an identifier. 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 as it reclaims unused memory.

## Array References

*Array references* are useful in several circumstances:

* 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

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

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

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

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

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`. Note the use of the scalar sigil (*sigils*) to access a single element.

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

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

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

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

To create an anonymous array--without using a declared array--surround a list of values with square brackets:

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

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

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

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

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

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

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

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

## Hash References

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

```
my %colors = (
    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>;
```

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

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

```
}
```

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

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

Create anonymous hashes in place with curly braces:

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

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

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

### Automatic Dereferencing

As of Perl 5.14, Perl can automatically dereference certain references on your behalf. Given an array reference in `$arrayref`, you can write:

```
push $arrayref, qw( list of values );
```

Given an expression which returns an array reference, you can do the same:

```
push $houses{$location}[$closets], \@new_shoes;
```

The same goes for the array operators `pop`, `shift`, `unshift`, `splice`, `keys`, `values`, and `each` and the hash operators `keys`, `values`, and `each`.

If the reference provided is not of the proper type--if it does not dereference properly--Perl will throw an exception. While this may seem more dangerous than explicitly dereferencing references directly, it is in fact the same behavior:

```
my $ref = sub { ... };

# will throw an exception
push  $ref, qw( list of values );

# will also throw an exception
push @$ref, qw( list of values );
```

### Function References

Perl 5 supports *first-class functions* in that a function is a data type just as is an array or hash. This is

most obvious with *function references*, and enables many advanced features (*closures*). 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;
```

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

Create anonymous functions with the bare `sub` keyword:

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

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

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

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

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

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

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

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

### Filehandle References

When you use `open`'s (and `opendir`'s) lexical filehandle form, you deal with filehandle references. Internally, these filehandles are `IO::File` objects. You can call methods on them directly. As of Perl 5.14, this is as simple as:

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

You must `use IO::File;` in 5.12 to enable this and `use IO::Handle;` in 5.10 and earlier. Even older code may take references to typeglobs:

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

This idiom predates lexical filehandles (introduced with Perl 5.6.0 in March 2000). You may still use the reference operator on typeglobs to take references to package-global filehandles such as `STDIN`,

STDOUT, STDERR, or DATA--but these are all global names anyhow.

Prefer lexical filehandles when possible. With the benefit of explicit scoping, lexical filehandles allow you to manage the lifespan of filehandles as a feature of Perl 5's memory management.

### Reference Counts

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

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

```
say 'file not open';

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

say 'file closed here';
```

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

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

### References and Functions

When you use references as arguments to functions, document your intent carefully. Modifying the values of a reference from within a function may surprise the calling code, which doesn't expect anything else to modify its data. To modify the contents of a reference without affecting the reference itself, copy its values to a new variable:

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

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

## Nested Data Structures

Perl's aggregate data types--arrays and hashes--allow you to store scalars indexed by integer or string keys. Perl 5's references (*references*) allow you to access aggregate data types 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.

Use the anonymous reference declaration syntax to declare a nested data structure:

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

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 as to ease adding new elements to the list.

Use Perl's reference syntax to access elements in nested data structures. 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};
```

The only way to nest a multi-level data structure is through references, so the arrow is superfluous. You may omit it for clarity, except for invoking function references:

```
my $nephew = $famous_triplets[1][2];
my $meal   = $meals{breakfast}{side};
$actions{financial}{buy_food}->( $nephew, $meal );
```

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

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

... or to slice a nested data structure:

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

Whitespace helps, but does not entirely eliminate the noise of this construct. Use temporary variables to clarify:

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

... or use `for`'s implicit aliasing to `$_` to avoid the use of an intermediate reference:

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

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

### Autovivification

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

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

After the second line of code, this array of arrays of arrays of arrays contains an array reference in an array reference in an array reference in an array reference. Each array reference contains one element. Similarly, treating an undefined value as if it were a hash reference in a nested data structure will make it so:

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

This useful behavior is *autovivification*. While it reduces the initialization code of nested data structures, it cannot distinguish between the honest intent to create missing elements in nested data structures and typos. The `autovivification` pragma (*pragmas*) from the CPAN lets you disable autovivification in a lexical scope for specific types of operations.

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 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 answers depend on your project. During early development, allow yourself the freedom to experiment. While testing and deploying, consider an increase of strictness to prevent unwanted side effects. Thanks to the lexical scoping of the `strict` and `autovivification` pragmas, you can enable these behaviors where and as necessary.

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

### Debugging Nested Data Structures

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

The core module `Data::Dumper` converts values of arbitrary complexity into strings of 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, its output is verbose. Some developers prefer the use of the `YAML::XS` or `JSON` modules for debugging. They do not produce Perl 5 code, 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 eventually point 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 zero or more children:

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

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

Both `$alice` and `$robert` contain an array reference which contains `$cianne`. Because `$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.

Either break the reference count manually yourself (by clearing the children of `$alice` and `$robert` or the parents of `$cianne`), or use *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`. Its `weaken()` function prevents a reference count from increasing:

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

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

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

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

Now `$cianne` will retain references to `$alice` and `$robert`, but those references will not by themselves prevent Perl's garbage collector from destroying those data structures. Most data structures do not need weak references, but when they're necessary, they're invaluable.

## Alternatives to Nested Data Structures

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