Standards and Styles for Developing Maintainable Code

Perl Best Practices

O'REILLY®

Damian Conway

CHAPTER 9
Subroutines

If you have a procedure with ten parameters,
you probably missed some.
—Alan Perlis

Subroutines are one of the two primary problem-decomposition tools available in Perl, modules being the other. They provide a convenient and familiar way to break a large task down into pieces that are small enough to understand, concise enough to implement, focused enough to test, and simple enough to debug.

In effect, subroutines allow programmers to extend the Perl language, creating useful new behaviours with sensible names. Having written a subroutine, you can immediately forget about its internals, and focus solely on the abstracted process or function it implements.

So the extensive use of subroutines helps to make a program more modular, which in turn makes it more robust and maintainable. Subroutines also make it possible to structure the actions of programs hierarchically, at increasingly high levels of abstraction, which improves the readability of the resulting code.

That's the theory, at least. In practice, there are plenty of ways that using subroutines can make code less robust, buggier, less concise, slower, and harder to understand. The guidelines in this chapter focus on avoiding those outcomes.

Call Syntax

Call subroutines with parentheses but without a leading &.

It's possible to call a subroutine without parentheses, if it has already been declared in the current namespace:

```
sub coerce;
```

```
# and later...

my $expected_count = coerce $input, $INTEGER, $ROUND_ZERO;
```

But that approach can quickly become much harder to understand:

```
fix my $gaze, upon each %suspect;
```

More importantly, leaving off the parentheses on subroutines makes them harder to distinguish from builtins, and therefore increases the mental search space when the reader is confronted with either type of construct. Your code will be easier to read and understand if the subroutines always use parentheses and the built-in functions always don't:

```
my $expected_count = coerce($input, $INTEGER, $ROUND_ZERO);

fix(my $gaze, upon(each %suspect));
```

Some programmers still prefer to call a subroutine using the ancient Perl 4 syntax, with an ampersand before the subroutine name:

```
&coerce($input, $INTEGER, $ROUND_ZERO);

&fix(my $gaze, &upon(each %suspect));
```

Perl 5 does support that syntax, but nowadays it's unnecessarily cluttered. Barewords is forbidden under `use strict`, so there are far fewer situations in which a subroutine call has to be disambiguated.

On the other hand, the ampersand itself is visually ambiguous; it can also signify a bitwise AND operator, depending on context. And context can be extremely subtle:

```
$curr_pos  = tell &get_mask();    # means: tell(get_mask())
$curr_time = time &get_mask();    # means: time() & get_mask()
```

Prefixing with `&` can also lead to other subtle (but radical) differences in behaviour:

```
sub fix {
    my (@args) = @_ ? @_ : $_;    # Default to fixing $_ if no args provided

    # Fix each argument by grammatically transforming it and then printing it...
    for my $arg (@args) {
        $arg =~ s/\A the \b/some/xms;
        $arg =~ s/e \z/es/xms;
        print $arg;
    }

    return;
}

# and later...

&fix('the race');    # Works as expected, prints: 'some races'
```

```
for ('the gaze', 'the adhesive') {
    &fix;           # Doesn't work as expected: looks like it should fix($_),
                    # but actually means fix(@_), using this scope's @_!
                    # See the 'perlsub' manpage for details
}
```

All in all, it's clearer, less ambiguous, and less error-prone to reserve the `&subname` syntax for taking references to named subroutines:

```
set_error_handler( \&log_error );
```

Just use parentheses to indicate a subroutine call:

```
coerce($input, $INTEGER, $ROUND_ZERO);

fix( my $gaze, upon(each %suspect) );

$curr_pos  = tell get_mask();
$curr_time = time & get_mask();
```

And *always* use the parentheses when calling a subroutine, even when the subroutine takes no arguments (like `get_mask( )`). That way it's immediately obvious that you intend a subroutine call:

```
curr_obj()->update($status);    # Call curr_obj() to get an object,
                                # then call the update()method on that object
```

and not a typename:

```
curr_obj->update($status);      # Maybe the same (if currobj( ) already declared),
                                # otherwise call update( ) on class 'curr_obj'
```

Homonyms

Don't give subroutines the same names as built-in functions.

If you declare a subroutine with the same name as a built-in function, subsequent invocations of that name will still call the builtin…except when occasionally they don't. For example:

```
sub lock {
    my ($file) = @_;
    return flock $file, LOCK_SH;
}

sub link {
    my ($text, $url) = @_;
    return qq{<a href="$url">$text</a>};
}

lock($file);                    # Calls 'lock' subroutine; built-in 'lock' hidden
print link($text, $text_url);   # Calls built-in 'link'; 'link' subroutine hidden
```

Perl considers some of its builtins (like `link`) to be "more built-in" than others (like `lock`), and chooses accordingly whether to call your subroutine of the same name. If the builtin is "strongly built-in", an ambiguous call will invoke it, in preference to any subroutine of the same name. On the other hand, if the builtin is "weakly built-in", an ambiguous call will invoke the subroutine of the same name instead.

Even if these subroutines did always work as expected, it's simply too hard to maintain code where the program-specific subroutines and the language's keywords overlap:

```
sub crypt { return "You're in the tomb of @_\n"   }
sub map   { return "You have found a map of @_\n" }
sub chop  { return "You have chopped @_\n"        }
sub close { return "The @_ is now closed\n"       }
sub hex   { return "A hex has been cast on @_\n"  }

print crypt( qw( Vlad Tsepes ) );             # Subroutine or builtin?

for my $reward (qw( treasure danger) ) {
     print map($reward, 'in', $location);     # Subroutine or builtin?
}

print hex('the Demon');                       # Subroutine or builtin?
print chop('the Demon');                      # Subroutine or builtin?
```

There is an inexhaustible supply of subroutine names available; names that are more descriptive and unambiguous. Use them:

```
sub in_crypt  { return "You're in the tomb of @_\n"   }
sub find_map  { return "You have found a map of @_\n" }
sub chop_at   { return "You have chopped @_\n"        }
sub close_the { return "The @_ is now closed\n"       }
sub hex_upon  { return "A hex has been cast on @_\n"  }

print in_crypt( qw( Vlad Tsepes ) );

for my $reward (qw( treasure danger )) {
    print find_map($reward, 'in', $location);
}

print hex_upon('the Demon');
print chop_at('the Demon');
```

Argument Lists

Always unpack `@_` first.

Subroutines always receive their arguments in the `@_` array. But accessing them via `$_[0]`, `$_[1]`, etc. directly is almost always a Very Bad Idea. For a start, it makes the code far less self-documenting:

```
Readonly my $SPACE => q{ };

# Pad a string with whitespace...
sub padded {
    # Compute the left and right indents required...
    my $gap   = $_[1] - length $_[0];
    my $left  = $_[2] ? int($gap/2) : 0;
    my $right = $gap - $left;

    # Insert that many spaces fore and aft...
    return $SPACE x $left
         . $_[0]
         . $SPACE x $right;
}
```

Using "numbered parameters" like this makes it difficult to determine what each argument is used for, whether they're being used in the correct order, and whether the computation they're used in is algorithmically sane. Compare the previous version to this one:

```
sub padded {
    my ($text, $cols_count, $want_centering) = @_;

    # Compute the left and right indents required...
    my $gap   = $cols_count - length $text;
    my $left  = $want_centering ? int($gap/2) : 0;
    my $right = $gap - $left;

    # Insert that many spaces fore and aft...
    return $SPACE x $left
         . $text
         . $SPACE x $right;
}
```

Here the first line unpacks the argument array to give each parameter a sensible name. In the process, that assignment also documents the expected order and intended purpose of each parameter. The sensible parameter names also make it easier to verify that the computation of `$left` and `$right` is correct.

A mistake when using numbered parameters:

```
my $gap   = $_[1] - length $_[2];
my $left  = $_[0] ? int($gap/2) : 0;
my $right = $gap - $left;
```

is much harder to identify than when named variables are in the wrong places:

```
my $gap   = $cols_count - length $want_centering;
my $left  = $text ? int($gap/2) : 0;
my $right = $gap - $left;
```

Moreover, it's easy to forget that each element of `@_` is an alias for the original argument; that changing `$_[0]` changes the variable containing that argument:

```
# Trim some text and put a "box" around it...
sub boxed {
    $_[0] =~ s{\A \s+ | \s+ \z}{}gxms;
    return "[$_[0]]";
}
```

Unpacking the argument list creates a copy, so it's far less likely that the original arguments will be inadvertently modified:

```
# Trim some text and put a "box" around it...
sub boxed {
    my ($text) = @_;

    $text =~ s{\A \s+ | \s+ \z}{}gxms;
    return "[$text]";
}
```

It's acceptable to unpack the argument list using a single list assignment as the first line of the subroutine:

```
sub padded {
    my ($text, $cols_count, $want_centering) = @_;

    # [Use parameters here, as before]
}
```

Alternatively, you can use a series of separate `shift` calls as the subroutine's first "paragraph":

```
sub padded {
    my $text           = shift;
    my $cols_count     = shift;
    my $want_centering = shift;

    # [Use parameters here, as before]
}
```

The list-assignment version is more concise, and it keeps the parameters together in a horizontal list, which enhances readability, provided that the number of parameters is small.

The `shift`-based version is preferable, though, whenever one or more arguments has to be sanity-checked or needs to be documented with a trailing comment:

```
sub padded {
    my $text           = _check_non_empty(shift);
    my $cols_count     = _limit_to_positive(shift);
    my $want_centering = shift;

    # [Use parameters here, as before]
}
```

Note the use of utility subroutines (see "Utility Subroutines" in Chapter 3) to perform the necessary argument verification and adjustment. Each such subroutine acts like a filter: it expects a single argument, checks it, and returns the argument value if the test succeeds. If the test fails, the verification subroutine may either return a default value instead, or call `croak( )` to throw an exception (see Chapter 13). Because of that second possibility, verification subroutines should be defined in the same package as the subroutines whose arguments they are checking.

This approach to argument verification produces very readable code, and scales well as the tests become more onerous. But it may be too expensive to use within small, frequently called subroutines, in which case the arguments should be unpacked in a list assignment and then tested directly:

```
sub padded {
    my ($text, $cols_count, $want_centering) = @_;
    croak  q{Can't pad undefined text}         if !defined $text;
    croak qq{Can't pad to $cols_count columns} if $cols_count <= 0;

    # [Use parameters here, as before]
}
```

The only circumstances in which leaving a subroutine's arguments in `@_` is appropriate is when the subroutine:

- Is short and simple
- Clearly doesn't modify its arguments in any way
- Only refers to its arguments collectively (i.e., doesn't index `@_`)
- Refers to `@_` only a small number of times (preferably once)
- Needs to be efficient

This is usually the case only in "wrapper" subroutines:

```
# Implement the Perl 6 print+newline function...
sub say {
    return print @_, "\n";
}

# and later...

say( 'Hello world!' );
say( 'Greetings to you, people of Earth!' );
```

In this example, copying the contents of `@_` to a lexical variable and then immediately passing those contents to `print` would be wasteful.

Named Arguments

Use a hash of named arguments for any subroutine that has more than three parameters.

Better still, use named arguments for any subroutine that is ever *likely* to have more than three parameters.

Named arguments replace the need to remember an ordering (which humans are comparatively poor at) with the need to remember names (which humans are relatively good at). Names are especially advantageous when a subroutine has many optional arguments—such as flags or configuration switches—only a few of which may be needed for any particular invocation.

Named arguments should always be passed to a subroutine inside a single hash, like so:

```
sub padded {
    my ($arg_ref) = @_;

    my $gap   = $arg_ref->{cols} - length $arg_ref->{text};
    my $left  = $arg_ref->{centered} ? int($gap/2) : 0;
    my $right = $gap - $left;

    return $arg_ref->{filler} x $left
           . $arg_ref->{text}
           . $arg_ref->{filler} x $right;
}

# and then...
for my $line (@lines) {
    $line = padded({ text=>$line, cols=>20, centered=>1, filler=>$SPACE });
}
```

As tempting as it may be, don't pass them as a list of raw name/value pairs:

```
sub padded {
    my %arg = @_;

    my $gap   = $arg{cols} - length $arg{text};
    my $left  = $arg{centered} ? int($gap/2) : 0;
    my $right = $gap - $left;

    return $arg{filler} x $left
           . $arg{text}
           . $arg{filler} x $right;
}
```

```
# and then...
for my $line (@lines) {
    $line = padded( text=>$line, cols=>20, centered=>1, filler=>$SPACE );
}
```

Requiring the named arguments to be specified inside a hash ensures that any mismatch, such as:

```
$line = padded({text=>$line, cols=>20..21, centered=>1, filler=>$SPACE});
```

will be reported (usually at compile time) in the caller's context:

```
Odd number of elements in anonymous hash at demo.pl line 42
```

Passing those arguments as raw pairs:

```
$line = padded(text=>$line, cols=>20..21, centered=>1, filler=>$SPACE);
```

would cause the exception to be thrown at run time, and from the line inside the subroutine where the odd number of arguments were unpacked and assigned to a hash:

```
Odd number of elements in hash assignment at Text/Manip.pm line 1876
```

It is okay to mix positional and named arguments, if there are always one or two main arguments to the subroutine (e.g., the string that `padded( )` is supposed to pad) and the remaining arguments are merely configuration options of some kind. In any case, when there are both positional arguments and named options, the unnamed positionals should come first, followed by a single reference to a hash containing the named options. For example:

```
sub padded {
    my ($text, $arg_ref) = @_;

    my $gap   = $arg_ref->{cols} - length $text;
    my $left  = $arg_ref->{centered} ? int($gap/2) : 0;
    my $right = $gap - $left;

    return $arg_ref->{filler} x $left . $text . $arg_ref->{filler} x $right;
}

# and then...
for my $line (@lines) {
    $line = padded( $line, {cols=>20, centered=>1, filler=>$SPACE} );
}
```

Note that using this approach also has a slight advantage in maintainability: it sets the options more clearly apart from the main positional argument.

By the way, you or your team might feel that three is not the most appropriate threshold for deciding to use named arguments, but try to avoid significantly larger values of "three". Most of the advantages of named arguments will be lost if you still have to plough through five or six positional arguments first.

Missing Arguments

Use definedness or existence to test for missing arguments.

It's a common mistake to use a boolean test to probe for missing arguments:

```
Readonly my $FILLED_USAGE => 'Usage: filled($text, $cols, $filler)';

sub filled {
    my ($text, $cols, $filler) = @_;

    croak $FILLED_USAGE
        if !$text || !$cols || !$filler;

    # [etc.]
}
```

The problem is that this approach can fail in subtle ways. If, for example, the filler character is `'0'` or the text to be padded is an empty string, then an exception will incorrectly be thrown.

A much more robust approach is to test for definedness:

```
use List::MoreUtils qw( any );

sub filled {
    my ($text, $cols, $filler) = @_;

    croak $FILLED_USAGE
        if any {!defined $_} $text, $cols, $filler;

    # [etc.]
}
```

Or, if a particular number of arguments is required, and `undef` is an acceptable value for one of them, test for mere existence:

```
sub filled {
    croak $FILLED_USAGE if @_ != 3;   # All three args must be supplied

    my ($text, $cols, $filler) = @_;
    # etc.
}
```

Existence tests are particularly efficient because they can be applied before the argument list is even unpacked. Testing for the existence of arguments also promotes more robust coding, in that it prevents callers from carelessly omitting a required argument, and from accidentally providing any extras.

Note that existence tests can also be used when some arguments are optional, because the recommended practice for this case—passing options in a hash—ensures

that the actual number of arguments passed is fixed (or fixed-minus-one, if the options hash happens to be omitted entirely):

```
sub filled {
    croak $FILLED_USAGE if @_ < 1 || @_ > 2;

    my ($text, $opt_ref) = @_;   # Cols and fill char now passed as options

    # etc.
}
```

Default Argument Values

Resolve any default argument values as soon as @_ is unpacked.

The fundamental rule of argument processing is: nothing happens in the subroutine until all the arguments are stable. Don't, for example, add in defaults on the fly:

```
Readonly my $DEF_PAGE_WIDTH => 78;
Readonly my $SPACE          => q{ };

sub padded {
    my ($text, $arg_ref) = @_;

    # Compute left and right spacings...
    my $gap   = ($arg_ref->{cols}||$DEF_PAGE_WIDTH) - length($text||=$EMPTY_STR);
    my $left  = $arg_ref->{centered} ? int($gap/2) : 0;
    my $right = $gap - $left;

    # Prepend and append space...
    my $filler = $arg_ref->{filler} || $SPACE;
    return $filler x $left . $text . $filler x $right;
}
```

Apart from making the gap computation much harder to read and to verify, using the `||` and `||=` operators to select default values is equivalent to testing for truth, and therefore much more prone to error on the edge cases (such as a `'0'` fill character).

If default values are needed, set them up first. Separating out any initialization will make your code more readable, and simplifying the computational statements is likely to make them less buggy too:

```
sub padded {
    my ($text, $arg_ref) = @_;

    # Set defaults...
    #           If option given...          Use option           Else default
    my $cols   = exists $arg_ref->{cols}   ? $arg_ref->{cols}   : $DEF_PAGE_WIDTH;
    my $filler = exists $arg_ref->{filler} ? $arg_ref->{filler} : $SPACE;
```

```
    # Compute left and right spacings...
    my $gap   = $cols - length $text;
    my $left  = $arg_ref->{centered} ? int($gap/2) : 0;
    my $right = $gap - $left;

    # Prepend and append space...
    return $filler x $left . $text . $filler x $right;
}
```

If there are many defaults to set up, the cleanest way to do that is by factoring the defaults out into a table (i.e., a hash) and then pre-initializing the argument hash with that table, like so:

```
Readonly my %PAD_DEFAULTS => (
    cols     => 78,
    centered => 0,
    filler   => $SPACE,
    # etc.
);

sub padded {
    my ($text, $arg_ref) = @_;

    # Unpack optional arguments and set defaults...
    my %arg = ref $arg_ref eq 'HASH' ? (%PAD_DEFAULTS, %{$arg_ref})
            :                          %PAD_DEFAULTS;

    # Compute left and right spacings...
    my $gap   = $arg{cols} - length $text;
    my $left  = $arg{centered} ? int($gap/2) : 0;
    my $right = $gap - $left;

    # Prepend and append space...
    return $arg{filler} x $left . $text . $arg{filler} x $right;
}
```

When the `%arg` hash is initialized, the defaults are placed ahead of the arguments supplied by the caller (`(%PAD_DEFAULTS, %{$arg_ref})`). So the entries in the default table are assigned to `%arg` first. Those default values are then overwritten by any entries from `$arg_ref`.

Scalar Return Values

Always `return scalar` in scalar returns.

One of the more subtle features of Perl subroutines is the way that their call context propagates to their `return` statements. In most places in Perl, the context (list, scalar, or void) can be deduced at compile time. One place where it *can't* be determined in advance is to the right of a `return`. The argument of a `return` is evaluated in whatever context the subroutine itself was called.

That's a very handy feature, which makes it easy to factor out or rename specific uses of built-in functions. For example, if you found yourself repeatedly filtering undefined and negative values out of lists:

```
@valid_samples = grep {defined($_) && $_ >= 0} @raw_samples;
```

it would be better to encapsulate that complex filter and rename it more meaningfully:

```
sub valid_samples_in {
    return grep {defined($_) && $_ >= 0} @_;
}

# and then...

@valid_samples = valid_samples_in(@raw_samples);
```

Because the `return` expression is always evaluated in the same context as the surrounding call, it's also still okay to use this subroutine in scalar context:

```
if (valid_samples_in(@raw_samples) < $MIN_SAMPLE_COUNT) {
    report_sensor_malfunction();
}
```

When the subroutine is called in scalar context, its `return` statement imposes scalar context on the `grep`, which then returns the total number of valid samples—just as a raw `grep` would do in the same position.

Unfortunately, it's easy to forget about the contextual lycanthropy of a `return`, especially when you write a subroutine that is "only ever going to be used one way"*. For example:

```
sub how_many_defined {
    return grep {defined $_} @_;
}

# and "always" thereafter:

my $found = how_many_defined(@raw_samples);
```

But eventually someone will write:

```
my ($found) = how_many_defined(@raw_samples);
```

and introduce a very subtle bug. The parentheses around `$found` put it in a list context, which puts the call to `how_many_defined( )` in a list context, which puts the `grep` inside `how_many_defined( )` in a list context, which causes the `return` to return the list of defined samples, the first of which is then assigned to `$found`†.

* Yep, that's the sound of alarm bells you're hearing.

† And if that sample happens to be an integer, then `$found` will be assigned a numeric value, exactly as expected. It will be the *wrong* numeric value, but hey, at least that will make the bug much more interesting to track down.

If there were even the slightest chance that this scalar-returning subroutine might ever be called in a list context, it should have been written as follows:

```
sub how_many_defined {
    return scalar grep {defined $_} @_;
}
```

There is no shame in using an explicit `scalar` anywhere you know you want a scalar but you're not confident of your context. And because you can never be confident of your context in a `return` statement, an explicit `scalar` is always acceptable there.

At very least, you should always add one anywhere that a previously mistaken expectation regarding context has already bitten you. That way, the same misconception won't bite whoever is eventually responsible for the care and feeding of your code (that is, most likely you again, six months later).

Contextual Return Values

Make list-returning subroutines return the "obvious" value in scalar context.

There is only one kind of list in Perl, so returning in a list context is easy—you just return all the values you produced:

```
sub defined_samples_in {
    return grep {defined $_} @_;
}
```

But what should that subroutine return in a scalar context? It might legitimately return an integer count (like `grep` itself does), in which case the subroutine stays exactly the same:

```
sub defined_samples_in {
    return grep {defined $_} @_;
}
```

Or it might instead return some serialized string representation of the list (like `localtime` does in scalar context):

```
sub defined_samples_in {
    my @defined_samples = grep {defined $_} @_;

    # Return all defined args in list context...
    if (wantarray) {
        return @defined_samples;
    }
    # Otherwise a serialized version in scalar context...
    return join($COMMA, @defined_samples);
}
```

Or it might return the "next" value in a series (like `readline` does):

```
use List::Util qw( first );

sub defined_samples_in {
    # Return all defined args in list context...
    if (wantarray) {
        return grep {defined $_} @_;
    }

    # Or, in scalar context, extract the first defined arg...
    return first {defined $_} @_;
}
```

It might try to preserve as much information as possible and return the full list of values using an array reference (which no Perl 5 builtin does):

```
sub defined_samples_in {
    my @defined_samples = grep {defined $_} @_;

    # Return all defined args in list context...
    if (wantarray) {
        return @defined_samples;
    }
    # Return all defined args (indirectly) in scalar context...
    return \@defined_samples;
}
```

It might even give up in disgust (like `sort` does):

```
sub defined_samples_in {
    croak q{Useless use of 'defined_samples_in' in a non-list context}
        if !wantarray;

    return grep {defined $_} @_;
}
```

Perl's list-returning builtins don't have a consistent behaviour in scalar context. They try to "do the right thing" on a case-by-case basis. Mostly they get it right; the scalar context results of `grep`, and `localtime`, and `readline` are what most people expect them to be.

Unfortunately, they don't *always* get it right. The scalar return values of `select`, `readpipe`, `splice`, `unpack`, and the various `get...` functions can be surprising to infrequent users of these functions. They have to be either memorized or repeatedly looked up in the fine manual. For many people, this makes using those builtins harder than it should be.

Don't perpetuate those difficulties in your own development. If you're writing a library of subroutines, make them predictable. Make every list-returning subroutine return the "obvious" value in scalar context.

What's the "obvious" value? It's the value that the developers who use the subroutine actually expect it to return. For example, if they all use `defined_samples_in( )` like so:

```
if ( defined_samples_in(@samples) > 0 ) {
    process(@samples);
}
```

then they obviously expect it to return a count of defined samples. So the "obvious" scalar context return value is that count.

On the other hand, if everyone uses it like this:

```
my $floor_samples_ref     = defined_samples_in(@floor_samples);
my $restocked_samples_ref = defined_samples_in(@restocked_samples);

# and later...

swap_arrays($floor_samples_ref, $restocked_samples_ref);
```

then the expectation is clearly that the subroutine returns a reference to the array of results. So *that's* the "obvious" scalar return value.

In other words, the "obvious" return value in a scalar context is whatever the people who use your code *think* it's going to be (before they read the fine manual). That definition of obviousness presents a dilemma, though. The way you work out whether your proposed scalar-context behaviour is obvious is by implementing it and seeing how many people it trips up. But once the subroutine is deployed and client code is relying on it, it's too late to change its return value if that value turns out not to be what most people expect.

The solution (which is discussed in greater detail in Chapter 17) is to "play test" the subroutine before it's deployed. That is, ask the people who will actually be using your subroutine what they expect it will do in scalar context. Or, better yet, have them write sample code that uses the subroutine, and see how they use it. If you get a consensus (or even just a simple majority opinion), implement that. If you don't get agreement on a single "obvious" behaviour, see the "Multi-Contextual Return Values" guideline later in this chapter.

Unfortunately, getting this kind of preliminary feedback isn't always feasible. In such cases, you should simply select a reasonable default, based on the three fundamental categories of list-returning subroutines: homogeneous, heterogeneous, and iterative.

A *homogeneous list-returning subroutine* is one that returns a list of data values that are all of a single type: a list of samples, a list of names, or a list of images. Perl's built-in `map`, `grep`, and `sort` are examples of this type of subroutine. Because no one value in a homogeneous list is more significant than any other, the only interesting property of the list in a scalar context is usually the number of values it contains. Hence, in scalar contexts, homogeneous subroutines are usually expected to return a count, as `map` and `grep` both do.

A *heterogeneous list-returning subroutine* is one that returns a list containing distinct pieces of information: name, rank, and serial number; account number, account name, and balance; year, month, day. For example, the `stat`, `caller`, and `getpwent` builtins are all heterogeneous. The lists returned by subroutines of this type often do have a single piece of information that is more significant than any other, and they're typically expected to return that value in scalar contexts. For example, `caller` returns the caller's package name, whilst `getpwent` returns the relevant username.

Alternatively, all of the information returned by a heterogeneous subroutine might be equally important. So this type of subroutine is sometimes expected to return some kind of serialized representation of that information in scalar context, as `localtime` and `gmtime` do.

An *iterative list-returning subroutine* is one that returns an iterated series of values, typically the result of successive input operations. The builtins `readline` and `readdir` work this way. Iterative subroutines are always used for stepping through sequences of data, so in a scalar context, they should always return the result of a single iteration.

Remember, though, that these suggested default behaviours are recommendations, not natural laws. You may find that your "play testing" suggests that some other return value is more appropriate—more expected—in your particular subroutine. In that case, you should implement and deploy that behaviour instead, and then explicitly document the reasons for your choice.

Multi-Contextual Return Values

When there is no "obvious" scalar context return value, consider `Contextual::Return` instead.

Sometimes no single scalar return value is appropriate for a list-returning subroutine. Your play-testers simply can't agree: different developers consistently expect different behaviours in different scalar contexts.

For example, suppose you're implementing a `get_server_status( )` subroutine that normally returns its information as a heterogeneous list:

```
# In list context, return all the available information...
my ($name, $uptime, $load, $users) = get_server_status($server_ID);
```

You may find that, in scalar contexts, some programmers expected it to return its numeric load value:

```
# Total load is sum of individual server loads...
$total_load += get_server_status($server_ID);
```

Others assumed it would return a boolean value indicating whether the server is up:

```
# Skip inactive servers...
next SERVER if ! get_server_status($server_ID);
```

Still others anticipated a string summarizing the current status:

```
# Compile report on all servers...
$servers_summary .= get_server_status($server_ID) . "\n";
```

While a fourth group hoped for a hash-reference, to give them convenient named access to the particular server information they wanted:

```
# Total users is sum of users on each server...
$total_users += get_server_status($server_ID)->{users};
```

In such cases, implementing any one of these four expectations is going to leave three-quarters of your developers unhappy.

At some point, every subroutine *will* be called in scalar context, and will have to return something. If that something isn't obvious to the majority of people, then inexperienced developers—who might not even realize their call is in scalar context—will suffer. And experienced developers will suffer too: ham-strung by the limitations of scalar context return and forced to work with your arbitrary choice of return value.

Perl's subroutines are context-sensitive for a reason: so that they can Do The Right Thing when used in different ways. But often in scalar contexts there is no one Right Thing. So developers give up and just pick the One Thing That Seems Rightest…to them. All too often, a decision like that leads to confusion, frustration, and buggy code.

Surprisingly, the underlying problem here isn't that Perl is context-sensitive. The problem is that Perl isn't context-sensitive *enough*.

Perl has one kind of list context and one kind of void context, so simple list-context and void-context returns are the perfect tools for those. On the other hand, Perl has at least a dozen distinct scalar subcontexts: boolean, integer, floating-point, string, and the numerous reference types. So, unless one of those return types is the clear and obvious candidate, simple scalar context return is totally inadequate: a sledgehammer when you really need tweezers.

Fortunately, there's a simple way to allow subroutines like `get_server_status()` to cater for two or more different scalar-context expectations simultaneously. The `Contextual::Return` CPAN module provides a mechanism by which you can specify that a subroutine returns different scalar values in boolean, numeric, string, hash-ref, array-ref, and code-ref contexts. For example, to allow `get_server_status()` to simultaneously support all five return behaviours shown at the start of this guideline, you could simply write:

```
use Contextual::Return;

sub get_server_status {
    my ($server_ID) = @_;
```

```
    # Acquire server data somehow...
    my %server_data
        = _ascertain_server_status($server_ID);

    # Return different components of that data, depending on call context...
    return (
        LIST    { @server_data{ qw( name uptime load users ) };                  }
        BOOL    { $server_data{uptime} > 0;                                      }
        NUM     { $server_data{load};                                            }
        STR     { "$server_data{name}: $server_data{uptime}, $server_data{load}"; }
        HASHREF { \%server_data;                                                 }
    );
}
```

Now, in a list context, `get_server_status()` uses a hash slice to extract the information in the expected order. In a boolean context, it returns true if the uptime is nonzero. In a numeric context, it returns the server load. In a string context, a string summarizing the server's status is returned. And when the return value is expected to be a hash reference, `get_server_status()` simply returns a reference to the entire `%server_data` hash.

Note that each of those alternative return values is lazily evaluated. That means, on any given call to `get_server_status()`, only one of the five contextual return blocks is actually executed.

Even in cases where you don't need to distinguish between so many alternatives, the `Contextual::Return` module can still improve the maintainability of your code, compared to using the built-in `wantarray`. The module allows you to say explicitly what you want to happen in different return context, and to label each of those outcomes with an obvious keyword. For example, suppose you had a subroutine such as:

```
sub defined_samples_in {
    if (wantarray) {
        return grep {defined $_} @_;
    }

    return first {defined $_} @_;
}
```

Without changing its behaviour at all, you could make the code considerably more self-documenting, and emphasize the inherent symmetry of the list and scalar cases, by rewriting it with a single contextual return:

```
use Contextual::Return;

sub defined_samples_in {
    return (
        LIST   {  grep {defined $_} @_ }
        SCALAR { first {defined $_} @_ }
    );
}
```

Besides producing more explicit and less cluttered code, this approach is more maintainable, too. When you need to extend the return behaviour of the subroutine, to more precisely match the expectations of those who use it, you can just add extra labeled return contexts, anywhere in the return list:

```
use Contextual::Return;

sub defined_samples_in {
    return (
        LIST     {        grep {defined $_} @_   }  # All defined vals
        SCALAR   {       first {defined $_} @_   }  # One defined val
        NUM      { scalar grep {defined $_} @_   }  # How many vals defined?
        ARRAYREF {      [ grep {defined $_} @_ ] }  # Return vals in an array
    );
}
```

Regardless of the order in which the alternatives appear, `Contextual::Return` will automatically select the most appropriate behaviour in each call context.

Prototypes

Don't use subroutine prototypes.

Subroutine prototypes allow you to make use of more sophisticated argument-passing mechanisms than Perl's "usual list-of-aliases" behaviour. For example:

```
sub swap_arrays (\@\@) {
    my ($array1_ref, $array2_ref) = @_;

    my @temp_array = @{$array1_ref};
    @{$array1_ref} = @{$array2_ref};
    @{$array2_ref} = @temp_array;

    return;
}

# and later...

swap_arrays(@sheep, @goats);      # Implicitly pass references
```

The problem is that anyone who uses `swap_arrays( )`, and anyone who subsequently has to maintain that code, has to know about that subroutine's special magic. Otherwise, they will quite naturally assume that the two arrays will be flattened into a single list and slurped up by the subroutine's `@_`, because that's what happens in just about every other subroutine they ever use.

Using prototypes makes it impossible to deduce the argument-passing behaviour of a subroutine call simply by looking at the call. They also make it impossible to deduce

the context in which particular arguments are evaluated. A subtle but common mistake is to "improve" the robustness of an existing library by putting prototype specifiers on all the subroutines. So a subroutine that used to be defined:

```
use List::Util qw( min max );

sub clip_to_range {
    my ($min, $max, @data) = @_;

    return map { max( $min, min($max, $_) ) } @data;
}
```

is updated to:

```
sub clip_to_range($$@) {  # takes two scalars and an array
    my ($min, $max, @data) = @_;

    return map { max($min, min($max, $_)) } @data;
}
```

The problem is that `clip_to_range( )` was being used with an elegant table-lookup scheme:

```
my %range = (
    normalized => [-0.5,0.5],
    greyscale  => [0,255],
    percentage => [0,100],
    weighted   => [0,1],
);

# and later...

my $range_ref = $range{$curr_range};
@samples = clip_to_range( @{$range_ref}, @samples);
```

The `$range{$curr_range}` hash look-up returns a reference to a two-element array corresponding to the range that's currently selected. That array reference is then dereferenced by putting a `@{...}` around it. Previously, when `clip_to_range( )` was an ordinary subroutine, that dereferenced array found itself in the list context, so it flattened into a list, producing the required minimum and maximum values for the subroutine's first two arguments.

But now that `clip_to_range( )` has a prototype, things go very wrong. The prototype starts with a `$`, which looks like it's telling Perl that the first argument must be a scalar. But that's not what prototypes do at all.

What that `$` prototype does is tell Perl that the first argument must be *evaluated in a scalar context*. And what is the first argument? It's the array produced by `@{$range{$curr_range}}`. And what do you get when an array is evaluated in a scalar context? The size of the array, which is 2, no matter which entry in `%range` was actually selected.

The second argument specification in the prototype is also a `$`. So the second argument to `clip_to_range( )` must also be evaluated in a scalar context. And that second argument? It's `@samples`. Evaluating that array in scalar context once again produces its size. The second argument becomes the number of samples.

The final specification in the prototype is a `@`, which specifies that any remaining arguments are evaluated in list context. Of course, there aren't any more arguments now, but the `@` specifier doesn't complain about that. An empty list is still a list, as far as it's concerned.

Adding a prototype didn't really improve the robustness of the code very much. Before it was imposed, `clip_to_range( )` would have been passed the selected minimum, followed by the selected maximum, followed by all the data samples. Now, thanks to the wonders of prototyping, `clip_to_range( )` always gets a minimum of 2, followed by a maximum equal to the number of samples, followed by no data. And Perl doesn't complain at all, since the prototype *was* successfully matched by the given arguments, even though it hosed them in the process.

Prototypes cause far more trouble than they avert. Even when they are properly understood and used correctly, they create code that doesn't behave the way it looks like it ought to, which makes it harder to maintain code that uses them. Furthermore, in OO implementations they engender a completely false sense of security, because they're utterly ignored in any method call.

Don't use prototypes. The only real advantage they can confer is allowing array and hash arguments to effectively be passed by reference:

```
swap_arrays(@sheep, @goats);
```

But even then, if you need pass-by-reference semantics, it's far better to make that explicit:

```
sub swap_arrays {
    my ($array1_ref, $array2_ref) = @_;

    my @temp_array = @{$array1_ref};
    @{$array1_ref} = @{$array2_ref};
    @{$array2_ref} = @temp_array;

    return;
}

# and later...

swap_arrays(\@sheep, \@goats);      # Explicitly pass references
```

Note that the body of `swap_arrays( )` shown here is exactly the same as in the prototyped version at the start of this guideline. Only the call syntax varies. With prototypes it's magical, and therefore misleading; without prototypes it's a little uglier, but shows at a glance exactly what the code is doing.

Implicit Returns

Always return via an explicit `return`.

If a subroutine "falls off the end" without ever encountering an explicit `return`, the value of the last expression evaluated in a subroutine is returned. That can lead to completely unexpected return values.

For example, consider this subroutine, which is supposed to return the second odd number in its argument list, or `undef` if there isn't a second odd number in the list:

```
sub find_second_odd {
    my $prev_odd_found = 0;

    # Check through args...
    for my $num (@_) {
        # Find an odd number...
        if (odd($num)) {
            # Return it if it's not the first (must be the second)...
            return $num if $prev_odd_found;

            # Otherwise, remember it's been seen...
            $prev_odd_found = 1;
        }
    }
    # Otherwise, fail
}
```

When that subroutine is used, strange things happen:

```
if (defined find_second_odd(2..6)) {
    # find_second_odd( ) returns 5
    # so the if block does execute as expected
}
if (defined find_second_odd(2..1)) {
    # find_second_odd( ) returns undef
    # so the if block is skipped as expected
}

if (defined find_second_odd(2..4)) {
    # find_second_odd( ) returns an empty string (!)
    # so the if block is unexpectedly executed
}

if (defined find_second_odd(2..3)) {
    # find_second_odd( ) returns an empty string again (!)
    # so the if block is unexpectedly executed again
}
```

The subroutine works correctly when there *is* a second odd number to be found, and when there are no numbers at all to be considered, but it behaves—there's no other

word for it—*oddly* for the in-between cases*. That anomalous empty string is returned because that's what a failed boolean test evaluates to in Perl. And a failed boolean test is the last expression evaluated in the loop. No, not the conditional in:

```
if (odd($num)) {
```

or in:

```
    return $num if $prev_found;
```

The last expression is the (failed) conditional test of the `while` loop. What `while` loop? The implicit `while` loop that the Perl compiler secretly translates every `for` loop into.

That's the problem. In order to predict the implicit return value of anything but the simplest subroutine, you not only have to understand the control flow within the subroutine and how that flow may change under different argument lists, but also what sneaky manipulations the compiler is performing on your code before it's executed.

But none of those complications will ever trouble you if you simply ensure that your subroutines can never "fall off the end". And all that requires is that every subroutine finishes with an explicit `return` statement—even if you have to add one "gratuitously":

```
sub find_second_odd {
    my $prev_odd_found = 0;

    # Check through args...
    for my $num (@_) {
        # Find an odd number...
        if (odd($num)) {
            # Return it if it's not the first (must be the second)...
            return $num if $prev_odd_found;

            # Otherwise, remember it's been seen...
            $prev_odd_found = 1;
        }
    }
    # Otherwise, fail explicitly
    return;
}
```

Now the subroutine always behaves as expected:

```
if (defined find_second_odd(2..6)) {
    # find_second_odd( ) returns 5
    # so if the block is executed, as expected
}
if (defined find_second_odd(2..1)) {
    # find_second_odd( ) returns undef
    # so if the block is skipped, as expected
}
```

* They'd be the "edge-cases", except that, in this instance, they're conceptually in the middle of the full range of possibilities.

```
if (defined find_second_odd(2..4)) {
    # find_second_odd( ) returns undef
    # so if the block is skipped, as expected
}

if (defined find_second_odd(2..3)) {
    # find_second_odd( ) returns undef
    # so if the block is skipped, as expected
}
```

That extra `return` is a very small price to pay for perfect predictability.

Note that this rule applies even if your subroutine "doesn't return anything". For example, if you're writing a subroutine to set a global flag, don't write:

```
sub set_terseness {
    my ($terseness) = @_;

    $default_terseness = $terseness;
}
```

If the subroutine isn't supposed to return a meaningful value, make it do so explicitly:

```
sub set_terseness {
    my ($terseness) = @_;

    $default_terseness = $terseness;

    return;  # Explicitly return nothing meaningful
}
```

Otherwise, developers who use the code could misinterpret the lack of an explicit `return` as indicating a deliberate implicit return instead. So they may come to rely on `set_terseness( )` returning the new terseness value. That misinterpretation will become a problem if you later realize that the subroutine actually ought to return the previous terseness value, because that change in behaviour will now break any client code that was previously relying on the "undocumented feature" provided by the implicit return.

Returning Failure

Use a bare `return` to return failure.

Notice that each final `return` statement in the examples of the previous guideline used a `return` keyword with no argument, rather than a more-explicit `return undef`.

Normally, relying on default behaviour is not best practice. But in the case of a `return` statement, relying on the default return value actually prevents a particularly nasty bug.

The problem with returning an explicit `return undef` is that—contrary to most people's expectations—a returned `undef` isn't always false.

Consider a simple subroutine like this:

```
use Contextual::Return;

sub guesstimate {
    my ($criterion) = @_;

    my @estimates;
    my $failed = 0;

    # [Acquire data for specified criterion]

    return undef if $failed;

    # [Do guesswork based on the acquired data]

    # Return all guesses in list context or average guess in scalar context...
    return (
        LIST   { @estimates                 }
        SCALAR { sum(@estimates)/@estimates; }
    );
}
```

The successful return values are both fine, and completely appropriate for the two contexts in which the subroutine might be called. But the failure value is a serious problem. Since `guesstimate( )` specifically tests for calls in list context, it's obvious that the subroutine is expected to be called in list contexts:

```
if (my @melt_rates = guesstimate('polar melting')) {
    my $model = Std::Climate::Model->new({ polar_melting => \@melt_rates });

    for my $interval (1,2,5,10,50,100,500) {
        print $model->predict({ year => $interval })
    }
}
```

But if the `guesstimate( )` subroutine fails, it returns a single scalar value: `undef`. And in a list context (such as the assignment to `@melt_rates`), that single scalar `undef` value becomes a one-element list: `(undef)`. So `@melt_rates` is assigned that one-element list and then evaluated in the overall *scalar* context of the `if` condition. And in scalar context an array always evaluates to the number of elements in it, in this case `1`. Which is true.

Oops![*]

What should have happened, of course, is that `guesstimate()` should have returned a failure value that was false in whatever context it was called, i.e., `undef` in scalar context and an empty list in list context:

```
if ($failed) {
    return (
        LIST   { ()    }
        SCALAR { undef }
    );
}
```

But that's precisely what a `return` itself does when it's not given an argument: it returns whatever the appropriate false value is for the current call context. So, by always using a bare `return` to return a "failure value", you can ensure that you will never bring about the destruction of the entire planetary ecosystem because of an expectedly true `undef`.

Meanwhile, Chapter 13 presents a deeper discussion on the most appropriate ways to propagate failure from a subroutine.

[*] And here "Oops!" means: the `if` block executes despite the failure of `guesstimate()` to acquire any meaningful data. So, when the climate model requests a numerical polar melting rate, that `undef` is silently converted to zero. This dwimmery causes the model to show that polar melting rates have absolutely no connection to world climate in general, and to rising ocean levels in particular. So mankind can happily keep burning fossil fuels at an ever-greater rate, secure in the knowledge that it has no effect. Until one day, the only person left is Kevin Costner. On a raft.