Abstract

Perl Cookbook is a comprehensive collection of problems, solutions, and practical examples for anyone programming in Perl. The book contains hundreds of rigorously reviewed Perl "recipes" and thousands of examples ranging from brief one-liners to complete applications.

Perl Best Practices is designed to help you write better Perl code: in fact, the best Perl code you possibly can. It's a collection of 256 guidelines covering various aspects of the art of coding, including layout, name selection, choice of data and control structures, program decomposition, interface design and implementation, modularity, object orientation, error handling, testing, and debugging. These guidelines have been developed and refined over a programming career spanning 22 years. They're designed to work well together, and to produce code that is clear, robust, efficient, maintainable, and concise.

Strings

Perl's fundamental unit for working with data is the scalar, that is, single values stored in single (scalar) variables. Scalar variables hold strings, numbers, and references. Array and hash variables hold lists or associations of scalars, respectively. References are used for referring to values indirectly, not unlike pointers in low-level languages. Numbers are usually stored in your machine's double-precision floating-point notation. Strings in Perl may be of any length, within the limits of your machine's virtual memory, and can hold any arbitrary data you care to put there—even binary data containing null bytes.

A string in Perl is not an array of characters—nor of bytes, for that matter. You cannot use array subscripting on a string to address one of its characters; use **substr** for that. Like all data types in Perl, strings grow on demand. Space is reclaimed by Perl's garbage collection system when no longer used, typically when the variables have gone out of scope or when the expression in which they were used has been evaluated. In other words, memory management is already taken care of, so you don't have to worry about it.

A scalar value is either defined or undefined. If defined, it may hold a string, number, or reference. The only undefined value is **undef**. All other values are defined, even numeric and the empty string. Definedness is not the same as Boolean truth, though; to check whether a value is defined, use the **defined** function. Boolean truth has a specialized meaning, tested with operators such as **&&** and **||** or in an **if** or **while** block's test condition.

Two defined strings are false: the empty string ("") and a string of length one containing the digit zero ("0"). All other defined values (e.g., "false", 15, and \$x) are true. You might be surprised to learn that "0" is false, but this is due to Perl's on-demand conversion between strings and numbers. The values 0., 0.00, and 0.0000000 are all numbers and are therefore false when unquoted, since the number zero in any of its guises is always false. However, those three values ("0.", "0.00", and "0.0000000") are true when used as literal quoted strings in your program code or when they're read from the command line, an environment variable, or an input file.

```
print "The value $n is ", $n ? "TRUE" : "FALSE", "\n";
That value 0.00000 is TRUE
print "The value $n is now ", $n ? "TRUE" : "FALSE", "\n";
That value 0 is now FALSE
```

The **undef** value behaves like the empty string ("") when used as a string, **0** when used as a number, and the null reference when used as a reference. But in all three possible cases, it's false.

Specify strings in your program using single quotes, double quotes, the quoting operators q// and qq//, or here documents. No matter which notation you use, string literals are one of two possible flavors: interpolated or uninterpolated. Interpolation governs whether variable references and special sequences are expanded. Most are interpolated by default, such as in patterns (/regex/) and running commands ($x = `cmd`).

Where special characters are recognized, preceding any special character with a backslash renders that character mundane; that is, it becomes a literal. This is often referred to as "escaping" or "backslash escaping."

Using single quotes is the canonical way to get an uninterpolated string literal. Three special sequences are still recognized: ' to terminate the string, **\'** to represent a single quote, and **** to represent a backslash in the string.

```
$string = '\n';                 # two characters, \ and an n
$string = 'Jon \'Maddog\' Orwant';  # literal single quotes
```

Double quotes interpolate variables and expand backslash escapes. These include "**\n**" (newline), "**\033**" (the character with octal value 33), "**\cJ**" (Ctrl-J), "**\x1B**" (the character with hex value 0x1B), and so on.

```
$string = "\n";                 # a "newline" character
$string = "Jon \"Maddog\" Orwant";  # literal double quotes
```

If there are no backslash escapes or variables to expand within the string, it makes no difference which flavor of quotes you use.

The q// and qq// quoting operators allow arbitrary delimiters on interpolated and uninterpolated literals, respectively, corresponding to single- and double-quoted strings. For an uninterpolated string literal that contains single quotes, it's easier to use q// than to escape all single quotes with backslashes:

```
$string = 'Jon \'Maddog\' Orwant';   # embedded single quotes
$string = q/Jon 'Maddog' Orwant/;    # same thing, but more legible
```

Choose the same character for both delimiters, as we just did with /, or pair any of the following four sets of bracketing characters:

```
$string = q[Jon 'Maddog' Orwant];   # literal single quotes
$string = q{Jon 'Maddog' Orwant};   # literal single quotes
$string = q(Jon 'Maddog' Orwant);   # literal single quotes
$string = q<Jon 'Maddog' Orwant>;   # literal single quotes
```

Here documents are a notation borrowed from the shell used to quote a large chunk of text. The text can be interpreted as single-quoted, double-quoted, or even as commands to be executed, depending on how you

quote the terminating identifier. Uninterpolated here documents do not expand the three backslash sequences the way single-quoted literals normally do. Here we double-quote two lines with a here document:

```
$a = <<"EOF";
This is a multiline here document
terminated by EOF on a line by itself
EOF
```

Notice there's no semicolon after the terminating EOF.

Unicode attempts to unify all character sets in the entire world, including many symbols and even fictional character sets. Under Unicode, different characters have different numeric codes, called *code points*.

Perl has supported Unicode since v5.6 or so, but it wasn't until the v5.8 release that Unicode support was generally considered robust and usable.

All Perl's string functions and operators, including those used for pattern matching, now operate on characters instead of octets.

Because characters with code points above 256 are supported, the chr function is no longer restricted to arguments under 256, nor is ord restricted to returning an integer smaller than that. Ask for chr(0x394), for example, and you'll get a Greek capital delta: Δ.

```
my $char = chr(0x394);
my $code = ord($char);
printf "char %s is code %d, %#04x\n", $char, $code, $code;

char Δ is code 916, 0x394
```

Certainly the internal representation requires more than just 8 bits for a numeric code that big. But you the programmer are dealing with characters as abstractions, not as physical octets. Low-level details like that are best left up to Perl.

You shouldn't think of characters and bytes as the same. Programmers who interchange bytes and characters are guilty of the same class of sin as C programmers who blithely interchange integers and pointers. Even though the underlying representations may happen to coincide on some platforms, this is just a coincidence, and conflating abstract interfaces with physical implementations will always come back to haunt you, eventually.

You have several ways to put Unicode characters into Perl literals. If you're lucky enough to have a text editor that lets you enter Unicode directly into your

Perl program, you can inform Perl you've done this via the use utf8 pragma. Another way is to use \x escapes in Perl interpolated strings to indicate a character by its code point in hex, as in \xC4. Characters with code points above 0xFF require more than two hex digits, so these must be enclosed in braces.

```
print "\xC4 and \x{0394} look different\n";

char Ä and∆ look different\n
```

Internally, Perl uses a format called UTF-8, but many other encoding formats for Unicode exist, and Perl can work with those, too. The use encoding pragma tells Perl in which encoding your script itself has been written, or which encoding the standard filehandles should use. The use open pragma can set encoding defaults for all handles. Special arguments to open or to binmode specify the encoding format for that particular handle. The -C command-line flag is a shortcut to set the encoding on all (or just standard) handles, plus the program arguments themselves. The environment variables PERLIO, PERL_ENCODING, and PERL_UNICODE all give Perl various sorts of hints related to these matters.

The substr function lets you read from and write to specific portions of the string.

```
$value = substr($string, $offset, $count);
$value = substr($string, $offset);

substr($string, $offset, $count) = $newstring;
substr($string, $offset, $count, $newstring);  # same as previous
substr($string, $offset)        = $newtail;
```

Strings are a basic data type; they aren't arrays of a basic data type. Instead of using array subscripting to access individual characters as you sometimes do in other programming languages, in Perl you use functions like unpack or substr to access individual characters or a portion of the string.

The offset argument to substr indicates the start of the substring you're interested in, counting from the front if positive and from the end if negative. If the offset is 0, the substring starts at the beginning. The count argument is the length of the substring.

```
my $first  = substr($string, 0, 1);  # "T"
my $start  = substr($string, 5, 2);  # "is"
my $rest   = substr($string, 13);    # "you have"
my $last   = substr($string, -1);    # "e"
my $end    = substr($string, -4);    # "have"
my $piece  = substr($string, -8, 3); # "you"
print $first, "\n";
```

```
print $start, "\n";
print $rest, "\n";
print $last, "\n";
print $end, "\n";
print $piece, "\n";
```

You can do more than just look at parts of the string with **substr**; you can actually change them. That's because **substr** is a particularly odd kind of function—an lvaluable one, that is, a function whose return value may be itself assigned a value.

```
$string = "This is what you have";
print $string;
This is what you have
substr($string, 5, 2) = "wasn't"; # change "is" to "wasn't"
This wasn't what you have
substr($string, -12)  = "ondrous";# "This wasn't wondrous"
This wasn't wondrous
substr($string, 0, 1) = "";      # delete first character
his wasn't wondrous
substr($string, -10)  = "";      # delete last 10 characters
his wasn'
```

Specify a format describing the layout of the record to unpack. For positioning, use lowercase "**x**" with a count to skip forward some number of bytes, an uppercase "**X**" with a count to skip backward some number of bytes, and an "@" to skip to an absolute byte offset within the record.

```
# extract column with unpack
my $a = "To be or not to be";
my $b = unpack("x6 A6", $a);  # skip 6, grab 6
print $b, "\n";
# => or not

($b, my $c) = unpack("x6 A2 X5 A2", $a); # forward 6, grab 2; backward 5, grab
2

print "$b\n$c\n";
# => or
# => be
```

You would like to supply a default value to a scalar variable, but only if it doesn't already have one.

```
# set $x to $y unless $x is already true
$x ||= $y;
```

If 0, "0", and "" are valid values for your variables, use defined instead:

```
# use $b if $b is defined, else $c
$a = defined($b) ? $b : $c;
```

The big difference between the two techniques (defined and ||) is what they test: definedness versus truth. Three defined values are still false in the world of Perl: 0, "0", and "". If your variable already held one of those, and you wanted to keep that value, a || wouldn't work. You'd have to use the more elaborate three-way test with defined instead. It's often convenient to arrange for your program to care about only true or false values, not defined or undefined ones.

Here's another example, which sets $dir to be either the first argument to the program or "/tmp" if no argument were given.

```
$dir = shift(@ARGV) || "/tmp";
```

We can do this without altering @ARGV:

```
$dir = $ARGV[0] || "/tmp";
```

If 0 is a valid value for $ARGV[0], we can't use ||, because it evaluates as false even though it's a value we want to accept. We must resort to Perl's only ternary operator, the ?: ("hook colon," or just "hook"):

```
$dir = defined($ARGV[0]) ? shift(@ARGV) : "/tmp";
```

Use list assignment to reorder the variables.

```
($VAR1, $VAR2) = ($VAR2, $VAR1);
```

You can even exchange more than two variables at once:

```
($alpha, $beta, $production) = ($beta, $production, $alpha);
```

Use ord to convert a character to a number, or use chr to convert a number to its corresponding character:

```
$num  = ord($char);
$char = chr($num);
```

The %c format used in printf and sprintf also converts a number to a character:

```
$char = sprintf("%c", $num);             # slower than chr($num)
printf("Number %d is character %c\n", $num, $num);
Number 101 is character e
```

Unlike low-level, typeless languages such as assembler, Perl doesn't treat characters and numbers interchangeably; it treats strings and numbers interchangeably. That means you can't just assign characters and numbers back and forth. Perl provides Pascal's **chr** and **ord** to convert between a character and its corresponding ordinal value.

```
@ascii_character_numbers = unpack("C*", "sample");
```

```
print "@ascii_character_numbers\n";
```

```
115 97 109 112 108 101
```

```
$word = pack("C*", @ascii_character_numbers);
$word = pack("C*", 115, 97, 109, 112, 108, 101);   # same
print "$word\n";
```

```
sample
```

```
@unicode_points = unpack("U*", "fac\x{0327}ade");
print "@unicode_points\n";
```

```
102 97 99 807 97 100 101
```

```
$word = pack("U*", @unicode_points);
print "$word\n";
```

```
façade
```

The **use charnames** pragma lets you use symbolic names for Unicode characters. These are compile-time constants that you access with the **\N{**CHARSPEC} double-quoted string sequence. Several subpragmas are supported. The **:full** subpragma grants access to the full range of character names, but you have to write them out in full, exactly as they occur in the Unicode character database, including the loud, all-capitals notation. The **:short** subpragma gives convenient shortcuts. Any import without a colon tag is taken to be a script name, giving case-sensitive shortcuts for those scripts.

```
use charnames ':full';
print "\N{GREEK CAPITAL LETTER DELTA} is called delta.\n";
```

```
Δ is called delta.
```

```
use charnames ':short';
print "\N{greek:Delta} is an upper-case delta.\n";
```

```
Δ is an upper-case delta.
```

Use split with a null pattern to break up the string into individual characters, or use unpack if you just want the characters' values:

```
@array = split(//, $string);     # each element a single character
@array = unpack("U*", $string);  # each element a code point (number)
```

Or extract each character in turn with a loop:

```
while (/(.)/g) {        # . is never a newline here
  # $1 has character, ord($1) its number
}
```

Use the reverse function in scalar context for flipping characters:

```
$revchars = reverse($string);
```

To flip words, use reverse in list context with split and join:

```
$revwords = join(" ", reverse split(" ", $string));
```

In a regular expression, the **\X** metacharacter matches an extended Unicode combining character sequence.

```
@chars = $string =~ /(\X)/g;
```

You want to convert tabs in a string to the appropriate number of spaces, or vice versa. Converting spaces into tabs can be used to reduce file size when the file has many consecutive spaces. Converting tabs into spaces may be required when producing output for devices that don't understand tabs or think them at different positions than you do.

```
use Text::Tabs;
my @lines_with_tabs = ("abcd\tcde\td");
my @expanded_lines  = expand(@lines_with_tabs);
print @expanded_lines, "\n";
my @tabulated_lines = unexpand(@expanded_lines);
print @tabulated_lines, "\n";
```

Use the lc and uc functions or the \L and \U string escapes.

```
$big = uc($little);          # "bo peep" -> "BO PEEP"
$little = lc($big);          # "JOHN"    -> "john"
$big = "\U$little";          # "bo peep" -> "BO PEEP"
$little = "\L$big";          # "JOHN"    -> "john"
```

To alter just one character, use the lcfirst and ucfirst functions or the \l and \u string escapes.

```
$big = "\u$little";          # "bo"      -> "Bo"
$little = "\l$big";          # "BoPeep"  -> "boPeep"
```

If you don't need it to be a scalar variable that can interpolate, the use constant pragma will work:

```
use constant AVOGADRO => 6.02252e23;
```

Numbers

Perl works hard to make life easy for you, and the facilities it provides for manipulating numbers are no exception to that rule. If you treat a scalar value as a number, Perl converts it to one.

Perl tries its best to interpret a string as a number when you use it as one (such as in a mathematical expression), but it has no direct way of reporting that a string doesn't represent a valid number. Perl quietly converts non-numeric strings to zero, and it will stop converting the string once it reaches a non-numeric character—so "**A7**" is still **0**, and "**7A**" is just **7**.

The CPAN module Regexp::Common provides a wealth of canned patterns that test whether a string looks like a number. Besides saving you from having to figure out the patterns on your own, it also makes your code more legible. By default, this module exports a hash called %RE that you index into, according to which kind of regular expression you're looking for. Be careful to use anchors as needed; otherwise, it will search for that pattern anywhere in the string. For example:

```
use Regexp::Common;
$string = "Gandalf departed from the Havens in 3021 TA.";
print "Is an integer\n"         if $string =~ / ^  $RE{num}{int}  $ /x;
print "Contains the integer $1\n" if $string =~ /   ( $RE{num}{int} )  /x;
```

The following examples are other patterns that the module can use to match numbers:

```
$RE{num}{int}{-sep=>',?'}            # match 1234567 or 1,234,567
$RE{num}{int}{-sep=>'.'}{-group=>4}   # match 1.2345.6789
$RE{num}{int}{-base => 8}            # match 014 but not 99
$RE{num}{int}{-sep=>','}{-group=3}    # match 1,234,594
$RE{num}{int}{-sep=>',?'}{-group=3}   # match 1,234 or 1234
$RE{num}{real}                      # match 123.456 or -0.123456
$RE{num}{roman}                      # match xvii or MCMXCVIII
```

Use Perl's hex function if you have a hexadecimal string like "2e" or "0x2e":

```
$number = hex($hexadecimal);        # hexadecimal only ("2e" becomes 47)
```

Use the oct function if you have a hexadecimal string like "0x2e", an octal string like "047", or a binary string like "0b101110":

```
$number = oct($hexadecimal);        # "0x2e" becomes 47
$number = oct($octal);             # "057"  becomes 47
$number = oct($binary);            # "0b101110" becomes 47
```

The **oct** function converts octal numbers with or without the leading "**0**".

You want to generate numbers that are more random than Perl's random numbers.

Use a different random number generator, such as those provided by the Math::Random and Math::TrulyRandom modules from CPAN:

```
use Math::TrulyRandom;
$random = truly_random_value();

use Math::Random;
$random = random_uniform();
```

Three useful functions for rounding floating-point values to integral ones are **int**, **ceil**, and **floor**. Built into Perl, **int** returns the integral portion of the floating-point number passed to it. This is called "rounding toward zero." This is also known as integer truncation because it ignores the fractional part: it rounds down for positive numbers and up for negative ones. The POSIX module's **floor** and **ceil** functions also ignore the fractional part, but they always round down and up to the next integer, respectively, no matter the sign.

```
use POSIX qw(floor ceil);

print floor(12.5), "\n";    # 12
print floor(12.4), "\n";    # 12
print ceil(12.5), "\n";     # 13
print ceil(12.4), "\n";     # 13
print int(12.5), "\n";      # 12
print int(12.4), "\n";      # 12
```

Date and Time

Perl's **time** function returns the number of seconds that have passed since the Epoch—more or less. POSIX requires that **time** not include leap seconds, a peculiar practice of adjusting the world's clock by a second here and there to account for the slowing down of the Earth's rotation due to tidal angular-momentum dissipation. To convert Epoch seconds into distinct values for days, months, years, hours, minutes, and seconds, use the **localtime** and **gmtime** functions.

Values (and their ranges) returned from localtime and gmtime

Variable	Values	Range
$sec	seconds	0-60
$min	minutes	0-59
$hours	hours	0-23
$mday	day of month	1-31
$mon	month of year	0-11, 0 = = January
$year	years since 1900	1-138 (or more)
$wday	day of week	0-6, 0 = = Sunday
$yday	day of year	0-365
$isdst	0 or 1	true if daylight saving is in effect

The values for seconds range from 0-60 to account for leap seconds; you never know when a spare second will leap into existence at the urging of various standards bodies.

In scalar context, localtime and gmtime return the date and time formatted as an ASCII string:

```
Fri Apr 11 09:27:08 1997
```

The standard Time::tm module provides a named interface to these values. The standard Time::localtime and Time::gmtime modules override the list-returning localtime and gmtime functions, replacing them with versions that return Time::tm objects. Compare these two pieces of code:

```
# using arrays
print "Today is day ", (localtime)[7], " of the current year.\n";
Today is day 117 of the current year.

# using Time::tm objects
use Time::localtime;
```

```
$tm = localtime;
print "Today is day ", $tm->yday, " of the current year.\n";
```

```
Today is day 117 of the current year.
```

To go from a list to Epoch seconds, use the standard Time::Local module. It provides the functions **timelocal** and **timegm**, both of which take a nine-element list and return an integer. The list's values have the same meaning and ranges as those returned by **localtime** and **gmtime**. The gmtime function works just as localtime does, but gives the answer in UTC instead of your local time zone.

Epoch seconds values are limited by the size of an integer. If you have a 32-bit signed integer holding your Epoch seconds, you can only represent dates (in UTC) from **Fri Dec 13 20:45:52 1901** to **Tue Jan 19 03:14:07 2038** (inclusive).

Use localtime, which returns values for the current date and time if given no arguments. You can either use localtime and extract the information you want from the list it returns:

```
my ($day, $month, $year) = ($tm->mday, $tm->mon, $tm->year);
print $day, "\n";
print $month, "\n";
print $year, "\n";
```

Use the timelocal or timegm functions in the standard Time::Local module, depending on whether the date and time is in the current time zone or in UTC.

```
use Time::Local;
$TIME = timelocal($sec, $min, $hours, $mday, $mon, $year);
$TIME = timegm($sec, $min, $hours, $mday, $mon, $year);
```

The built-in function **localtime** converts an Epoch seconds value to distinct DMYHMS values; the **timelocal** subroutine from the standard Time::Local module converts distinct DMYHMS values to an Epoch seconds value.

Arrays

You can't use nested parentheses to create a list of lists. If you try that in Perl, your lists get *flattened*, meaning that both these lines are equivalent:

```
@nested = ("this", "that", "the", "other");
@nested = ("this", "that", ("the", "other"));
```

If you have a lot of single-word elements, use the qw( ) operator:

```
@a = qw(Meddle not in the affairs of wizards.);
```

The push function is optimized for appending a list to the end of an array. You can take advantage of Perl's list flattening to join two arrays, but this results in significantly more copying than push:

```
@ARRAY1 = (@ARRAY1, @ARRAY2);
```

If you're using reverse to reverse a list that you just sorted, you should have sorted it in the correct order to begin with. For example:

```
# two-step: sort then reverse
@ascending = sort { $a cmp $b } @users;
@descending = reverse @ascending;

# one-step: sort with reverse comparison
@descending = sort { $b cmp $a } @users;
```

The List::Util module, shipped standard with Perl as of v5.8 but available on CPAN for earlier versions, provides an even easier approach:

```
use List::Util qw(first);
$match = first { CRITERION } @list

my @list = (1, 2, 3, 4, 5);
my $match = first { $_ % 2 == 1 } @list;

print $match, "\n";
```

Use grep to apply a condition to all elements in the list and return only those for which the condition was true. The Perl **grep** function is shorthand for all that looping and mucking about. It's not really like the Unix **grep** command; it doesn't have options to return line numbers or to negate the test, and it isn't limited to regular-expression tests.

```
@MATCHING = grep { TEST ($_) } @LIST;
```

The sort function takes an optional code block, which lets you replace the default alphabetic comparison with your own subroutine. This comparison

function is called each time sort has to compare two values. The values to compare are loaded into the special package variables $a and $b, which are automatically localized.

The comparison function should return a negative number if $a ought to appear before $b in the output list, 0 if they're the same and their order doesn't matter, or a positive number if $a ought to appear after $b. Perl has two operators that behave this way: <=> for sorting numbers in ascending numeric order, and cmp for sorting strings in ascending alphabetic order. By default, sort uses cmp-style comparisons.

You want to sort a list by something more complex than a simple string or numeric comparison.

You can speed this up by precomputing the field.

```
@precomputed = map { [compute( ),$_] } @unordered;
@ordered_precomputed = sort { $a->[0] <=> $b->[0] } @precomputed;
@ordered = map { $_->[1] } @ordered_precomputed;
```

And, finally, you can combine the three steps:

```
@ordered = map { $_->[1] }
           sort { $a->[0] <=> $b->[0] }
           map { [compute( ), $_] }
           @unordered;
```

We can put multiple comparisons in the routine and separate them with ||. || is a short-circuit operator: it returns the first true value it finds. This means we can sort by one kind of comparison, but if the elements are equal (the comparison returns 0), we can sort by another. This has the effect of a sort within a sort:

```
@sorted = sort { $a->name cmp $b->name
                           ||
                 $b->age <=> $a->age } @employees;
```

Let's apply map-sort-map to the sorting by string length example:

```
@temp   = map  {  [ length $_, $_ ]  } @strings;
@temp   = sort { $a->[0] <=> $b->[0] } @temp;
@sorted = map  {      $_->[1]     } @temp;
```

We can combine it into one statement and eliminate the temporary array:

```
@sorted = map { $_->[1] }
          sort { $a->[0] <=> $b->[0] }
          map { [ length $_, $_ ] }
          @strings;
```

Use the **shuffle** function from the standard List::Util module, which returns the elements of its input list in a random order.

```
use List::Util qw(shuffle);
@array = shuffle(@array);
```

Use the appropriate functions from the standard Hash::Util module.

```
use Hash::Util qw{ lock_keys  unlock_keys
                   lock_value unlock_value
                   lock_hash  unlock_hash  };
```

To restrict access to keys already in the hash, so no new keys can be introduced:

```
lock_keys(%hash);              # restrict to current keys
lock_keys(%hash, @klist);      # restrict to keys from @klist
```

To forbid deletion of the key or modification of its value:

```
lock_value(%hash, $key);
```

To make all keys and their values read-only:

```
lock_hash(%hash);
```

The **delete** function is the only way to remove a specific entry from a hash. Once you've deleted a key, it no longer shows up in a **keys** list or an **each** iteration, and **exists** will return false for that key.

Use **reverse** to create an inverted hash whose values are the original hash's keys and vice versa.

Central to file access in Perl is the filehandle, like **INPUT** in the previous code example. Filehandles are symbols inside your Perl program that you associate with an external file, usually using the **open** function. Whenever your program performs an input or output operation, it provides that operation with an internal filehandle, not an external filename. It's the job of **open** to make that association, and of **close** to break it.

While users think of open files in terms of those files' names, Perl programs do so using their filehandles. But as far as the operating system itself is concerned, an open file is nothing more than a *file descriptor*, which is a small, non-negative integer.

Packages, Libraries, and Modules

Unlike user-defined identifiers, built-in variables with punctuation names (like **$_** and **$.**) and the identifiers **STDIN**, **STDOUT**, **STDERR**, **ARGV**, **ARGVOUT**, **ENV**, **INC**, and **SIG** are all forced to be in package **main** when unqualified.

The unit of software reuse in Perl is the module, a file containing related functions designed to be used by programs and other modules. Every module has a public interface, a set of variables and functions that outsiders are encouraged to use. From inside the module, the interface is defined by initializing certain package variables that the standard Exporter module looks at. From outside the module, the interface is accessed by importing symbols as a side effect of the **use** statement. The public interface of a Perl module is whatever is documented to be public.

The require and use statements load a module into your program, although their semantics vary slightly. require loads modules at runtime, with a check to avoid the redundant loading of a given module. use is like require, with two added properties: compile-time loading and automatic importing.

Modules included with **use** are processed at compile time, but **require** processing happens at runtime. This is important because if a module needed by a program is missing, the program won't even start because the **use** fails during compilation of your script. Another advantage of compile-time **use** over runtime **require** is that function prototypes in the module's subroutines become visible to the compiler. This matters because only the compiler cares about prototypes, not the interpreter.

The other difference between **require** and **use** is that **use** performs an implicit import on the included module's package. Importing a function or variable from one package to another is a form of aliasing; that is, it makes two different names for the same underlying thing. It's like linking files from another directory into your current one by the command ln /somedir/somefile. Once it's linked in, you no longer have to use the full pathname to access the file. Likewise, an imported symbol no longer needs to be fully qualified by package name (or declared with **our** or the older **use vars** if a variable, or with **use subs** if a subroutine). You can use imported variables as though they were part of your package. If you imported **$English::OUTPUT_AUTOFLUSH** in the current package, you could refer to it as **$OUTPUT_AUTOFLUSH**.

If the module name itself contains any double colons, these are translated into your system's directory separator. That means that the File::Find module resides in the file File/Find.pm under most filesystems. For example:

```
require "FileHandle.pm";        # runtime load
require FileHandle;             # ".pm" assumed; same as previous
use FileHandle;                 # compile-time load
```

```
require "Cards/Poker.pm";       # runtime load
require Cards::Poker;           # ".pm" assumed; same as previous
use Cards::Poker;               # compile-time load
```

The following is a typical setup for a hypothetical module named Cards::Poker that demonstrates how to manage its exports. The code goes in the file named Poker.pm within the directory Cards; that is, Cards/Poker.pm.

```
1   package Cards::Poker;
2   use Exporter;
3   @ISA = ("Exporter");
4   @EXPORT = qw(&shuffle @card_deck);
5   @card_deck = ( );               # initialize package global
6   sub shuffle { }                 # fill-in definition later
7   1;                              # don't forget this
```

In module file YourModule.pm, place the following code. Fill in the ellipses as explained in the Discussion section.

```
package YourModule;
use strict;
our (@ISA, @EXPORT, @EXPORT_OK, %EXPORT_TAGS, $VERSION);

use Exporter;
$VERSION = 1.00;            # Or higher
@ISA = qw(Exporter);

@EXPORT     = qw(...);      # Symbols to autoexport (:DEFAULT tag)
@EXPORT_OK  = qw(...);      # Symbols to export on request
%EXPORT_TAGS = (            # Define names for sets of symbols
    TAG1 => [...],
    TAG2 => [...],
    ...
);

##########################
# your code goes here
##########################

1;                          # this should be your last line
```

In other files where you want to use YourModule, choose one of these lines:

```
use YourModule;             # Import default symbols into my package
use YourModule qw(...);     # Import listed symbols into my package
use YourModule ( );         # Do not import any symbols
use YourModule qw(:TAG1);   # Import whole tag set
```

$VERSION

When a module is loaded, a minimal required version number can be supplied. If the version isn't at least this high, the use will raise an exception.

```
use YourModule 1.86;      # If $VERSION < 1.86, fail
```

@EXPORT

This array contains a list of functions and variables that will be exported into the caller's own namespace so they can be accessed without being fully qualified. Typically, a qw( ) list is used.

```
@EXPORT = qw(&F1 &F2 @List);
@EXPORT = qw( F1  F2 @List);      # same thing
```

To load the module at compile time but request that no symbols be exported, use the special form use Exporter ( ), with empty parentheses.

@EXPORT_OK

This array contains symbols that can be imported if they're specifically asked for. If the array were loaded this way:

```
@EXPORT_OK = qw(Op_Func %Table);
```

then the user could load the module like so:

```
use YourModule qw(Op_Func %Table F1);
```

and import only the Op_Func function, the %Table hash, and the F1 function. The F1 function was listed in the @EXPORT array. Notice that this does not automatically import F2 or @List, even though they're in @EXPORT. To get everything in @EXPORT plus extras from @EXPORT_OK, use the special :DEFAULT tag, such as:

```
use YourModule qw(:DEFAULT %Table);
```

%EXPORT_TAGS

This hash is used by large modules like CGI or POSIX to create higher-level groupings of related import symbols. Its values are references to arrays of symbol names, all of which must be in either @EXPORT or @EXPORT_OK. Here's a sample initialization:

```
%EXPORT_TAGS = (
     Functions => [ qw(F1 F2 Op_Func) ],
     Variables => [ qw(@List %Table)  ],
);
```

An import symbol with a leading colon means to import a whole group of symbols. Here's an example:

```
use YourModule qw(:Functions %Table);
```

That pulls in all symbols from:

```
@{ $YourModule::EXPORT_TAGS{Functions} },
```

that is, it pulls in the F1, F2, and Op_Func functions and then the %Table hash.

Although you don't list it in %EXPORT_TAGS, the implicit tag :DEFAULT automatically means everything in @EXPORT.

You need to load in a module that might not be present on your system. This normally results in a fatal exception. You want to detect and trap these failures.

Wrap the require or use in an eval, and wrap the eval in a BEGIN block:

```
# no import
BEGIN {
    unless (eval "require $mod; 1") {
        warn "couldn't require $mod: $@";
    }
}
```

```
# imports into current package
BEGIN {
    unless (eval "use $mod; 1") {
        warn "couldn't use $mod: $@";
    }
}
```

Programs that check their arguments and abort with a usage message on error have no reason to load modules they never use. This delays the inevitable and annoys users. But those use statements happen during compilation, not execution, as explained in the Introduction.

Here, an effective strategy is to place argument checking in a BEGIN block before loading the modules. The following is the start of a program that checks to make sure it was called with exactly two arguments, which must be whole numbers, before going on to load the modules it will need:

```
BEGIN {
    unless (@ARGV == 2 && (2 == grep {/^\d+$/} @ARGV)) {
        die "usage: $0 num1 num2\n";
    }
}
use Some::Module;
use More::Modules;
```

To find the current package:

```
$this_pack = _ _PACKAGE_ _;
```

To find the caller's package:

```
$that_pack = caller( );
```

The _ _**PACKAGE**_ _ symbol returns the package that the code is currently being compiled into. This doesn't interpolate into double-quoted strings.

END routines work like exit handlers, such as **trap 0** in the shell, **atexit** in C programming, or global destructors or finalizers in object-oriented languages. All of the ENDs in a program are run in the opposite order that they were loaded;

that is, last seen, first run.

You want to prepare your module in standard distribution format so you can easily send your module to a friend. It's best to start with Perl's standard h2xs tool.

```
% h2xs -XA -n Planets
% h2xs -XA -n Astronomy::Orbits
```

These commands make subdirectories called ./Planets/ and ./Astronomy/Orbits/, respectively, where you will find all the components you need to get you started. The -n flag names the module you want to make, -X suppresses creation of XS (external subroutine) components, and -A means the module won't use the AutoLoader.

You can get a quick start on writing modules using the h2xs program. This tool gives you a skeletal module file with the right parts filled in, and it also gives you the other files needed to correctly install your module and its documentation or to bundle up for contributing to CPAN or sending off to a friend.

If you plan to use autoloading, omit the -A flag to h2xs, which produces lines like this:

```
require Exporter;
require AutoLoader;
@ISA = qw(Exporter AutoLoader);
```

If your module is bilingual in Perl and C, omit the -X flag to h2xs to produce lines like this:

```
require Exporter;
require DynaLoader;
@ISA = qw(Exporter DynaLoader);
```

When you load a module using require or use, the entire module file must be read and compiled (into internal parse trees, not into byte code or native machine code) right then. For very large modules, this annoying delay is unnecessary if you need only a few functions from a particular file.

To address this problem, the SelfLoader module delays compilation of each subroutine until that subroutine is actually called. SelfLoader is easy to use: just place your module's subroutines underneath the _ _DATA_ _ marker so the compiler will ignore them, use a require to pull in the SelfLoader, and include SelfLoader in the module's @ISA array. That's all there is to it. When your module is loaded, the SelfLoader creates stub functions for all routines below _ _DATA_ _. The first time a function gets called, the stub replaces itself by first compiling the real function and then calling it.

```
require Exporter;
require SelfLoader;
@ISA = qw(Exporter SelfLoader);
#
```

```
    # other initialization or declarations here
    #
    _ _DATA_ _
    sub abc { .... }
    sub def { .... }
```

You'd like to write functions in C that you can call from Perl. You may already have tried XS and found it harmful to your mental health.

Use the Inline::C module available from CPAN:

```
    use Inline C;
    my $answer = somefunc(20, 4);
    print "$answer\n";             # prints 80
    _ _END_ _
    _ _C_ _
    double somefunc(int a, int b) {
        double answer = a * b;
        return answer;
    }
```

Inline::C was created as an alternative to the XS system for building C extension modules. Rather than jumping through all the hoopla of *h2xs* and the format of an *.xs* file, Inline::C lets you embed C code into your Perl program. There are also Inline modules for Python, Ruby, and Java, among other languages.

By default, your C source is in the _ _END_ _ or _ _DATA_ _ section of your program after a _ _C_ _ token. This permits multiple Inlined language blocks in a single file. If you want, use a here document when you load Inline:

```
    use Inline C <<'END_OF_C';
    double somefunc(int a, int b) {  /* Inline knows most basic C types */
        double answer = a * b;
        return answer;
    }
    END_OF_C
```

Inline::C scans the source code for ANSI-style function definitions. When it finds a function definition it knows how to deal with, it creates a Perl wrapper for the function.

Subroutines

All incoming parameters appear as separate scalar values in the special array @_, which is automatically local to each function. To return a value or values from a subroutine, use the **return** statement with arguments. If there is no **return** statement, the return value is the result of the last evaluated expression.

The scalars in @_ are implicit aliases for the ones passed in, not copies. That means changing the elements of @_ in a subroutine changes the values in the subroutine's caller. This is a holdover from before Perl had proper references.

The my operator confines a variable to a particular region of code in which it can be used and accessed. Outside that region, it can't be accessed. This region is called its scope. Variables declared with **my** have lexical scope, meaning that they exist only within a specific textual region of code.

Code can always determine the current source line number via the special symbol __LINE__, the current file via __FILE__, and the current package via __PACKAGE__. But no such symbol for the current subroutine name exists, let alone the name for the subroutine that called this one.

The built-in function caller handles all of these. In scalar context it returns the calling function's package name, but in list context it returns much more. You can also pass it a number indicating how many frames (nested subroutine calls) back you'd like information about: 0 is your own function, 1 is your caller, and so on.

Here's the full syntax, where $i is how far back you're interested in:

```
($package, $filename, $line, $subr, $has_args, $wantarray
#   0        1          2      3        4         5
 $evaltext, $is_require, $hints, $bitmask
#   6         7            8        9
) = caller($i);
```

Here's what each of those return values means:

$package

The package in which the code was compiled.

$filename

The name of the file in which the code was compiled, reporting -e if launched from that command-line switch, or - if the script was read from standard input.

$line

The line number from which that frame was called.

$subr

The name of that frame's function, including its package. Closures are indicated by names like main::__ANON__, which are not callable. In an eval, it contains (eval).

$has_args

Whether the function had its own @_ variable set up. It may be that there are no arguments, even if true. The only way for this to be false is if the function was called using the &fn notation instead of fn( ) or &fn( ).

$wantarray

The value the wantarray function would return for that stack frame; either true, false but defined, or else undefined. This tells whether the function was called in list, scalar, or void context (respectively).

$evaltext

The text of the current eval STRING, if any.

$is_require

Whether the code is currently being loaded by a do, require, or use.

$hints, $bitmask

These both contain pragmatic hints that the caller was compiled with. Consider them to be for internal use only by Perl itself.

Rather than using caller directly as in the Solution, you might want to write functions instead:

You want to know in which context your function was called. Use the wantarray( ) function, which has three possible return values, depending on how the current function was called:

```
if (wantarray()) {
    # list context
}
elsif (defined wantarray()) {
    # scalar context
}
else {
    # void context
}
```

Many built-in functions act differently when called in scalar context than they do when called in list context. A user-defined function can learn which context it was called in by checking wantarray. List context is indicated by a true return value. If wantarray returns a value that is false but defined, then the function's return value will be used in scalar context. If wantarray returns undef, your function isn't being asked to provide any value at all.

You want to make a function with many parameters that are easy to call so that programmers remember what the arguments do, rather than having to memorize their order.

```
sub thefunc {
    my %args = (
        INCREMENT     => '10s',
        FINISH        => 0,
```

```
        START        => 0,
        @_,
    );
    print $args{INCREMENT}, "\n";
    print $args{START}, "\n";
    print $args{FINISH}, "\n";
}

thefunc(INCREMENT => "20s", START => "+5m", FINISH => "+30m");
thefunc(INCREMENT => "20s", START => "+5m");
thefunc(START => "+5m");
```

More flexible approach allows the caller to supply arguments using name-value pairs. The first element of each pair is the argument name; the second, its value. This makes for self-documenting code because you can see the parameters' intended meanings without having to read the full function definition. Even better, programmers using your function no longer have to remember argument order, and they can leave unspecified any extraneous, unused arguments.

This works by having the function declare a private hash variable to hold the default parameter values. Put the current arguments, @_, after the default values, so the actual arguments override the defaults because of the order of the values in the assignment.

You have a function that returns many values, but you only care about some of them.

Either assign to a list that has undef in some positions:

```
($a, undef, $c) = func();
```

or else take a slice of the return list, selecting only what you want:

```
($a, $c) = (func())[0,2];
```

You want to return a value indicating that your function failed. Use a bare return statement without any argument, which returns undef in scalar context and the empty list () in list context. A return without an argument means:

```
sub empty_retval {
    return ( wantarray ? () : undef );
}
```

Manually checking the validity of a function's arguments can't happen until runtime. If you make sure the function is declared before it is used, you can tickle the compiler into using a very limited form of prototype checking. But don't confuse Perl's function prototypes with those found in any other language.

A Perl function prototype is zero or more spaces, backslashes, or type characters enclosed in parentheses after the subroutine definition or name. A backslashed type symbol means that the argument is passed by reference, and

the argument in that position must start with that type character.

A prototype can impose context on the prototyped function's arguments. This is done when Perl compiles your program. But this does not always mean that Perl checks the number or type of arguments; since a scalar prototype is like inserting a scalar in front of just one argument, sometimes an implicit conversion occurs instead. For example, if Perl sees func(3, 5) for a function prototyped as sub func ($), it will stop with a compile-time error. But if it sees func(@array) with the same prototype, it will merely put @array into scalar context instead of complaining that you passed an array, but it wanted a scalar.

This is so important that it bears repeating: don't use Perl prototypes expecting the compiler to check type and number of arguments for you. It does a little bit of that, sometimes, but mostly it's about helping you type less, and sometimes to emulate the calling and parsing conventions of built-in functions.

```
sub testfun(\@$\%) {
    my ($ary_ref, $arg1, $hash_ref) = @_;
    print $ary_ref->[1], "\n";
    print $arg1, "\n";
    print $hash_ref->{FINISH}, "\n";
}

my @a = (1, 2, 3);
my %h = (INCREMENT => "20s", START => "+5m", FINISH => "+30m");
testfun(@a, "123", %h);
```

Sometimes you encounter a problem so exceptional that merely returning an error isn't strong enough, because the caller could unintentionally ignore the error. Use die STRING from your function to trigger an exception:

```
die "some message";       # raise exception
```

The caller can wrap the function call in an eval to intercept that exception, then consult the special variable $@ to see what happened:

```
eval { func() };
if ($@) {
    warn "func raised an exception: $@";
}
```

To detect this, wrap the call to the function with a block eval. The $@ variable will be set to the offending exception if one occurred; otherwise, it will be false.

```
eval { $val = func() };
warn "func blew up: $@" if $@;
```

Use the local operator to save a previous global value, automatically restoring it when the current block exits:

```
our $age = 18;        # declare and set global variable
```

```
    if (CONDITION) {
        local $age = 23;
        func();       # sees temporary value of 23
    } # Perl restores the old value at block exit
```

Despite its name, Perl's **local** operator does not create a local variable. That's what **my** does. Instead, **local** merely preserves an existing value for the duration of its enclosing block. Hindsight shows that if **local** had been called *save_value* instead, much confusion could have been avoided.

Three places where you must use **local** instead of **my** are:

1. You need to give a global variable a temporary value, especially **$_**.
2. You need to create a local file or directory handle or a local function.
3. You want to temporarily change just one element of an array or hash.

Although a lot of old code uses local, it's definitely something to steer clear of when it can be avoided. Because local still manipulates the values of global variables, not local variables, you'll run afoul of use strict unless you declared the globals using our or the older use vars.

The local operator produces *dynamic scoping* or *runtime scoping*. This is in contrast with the other kind of scoping Perl supports, which is much more easily understood. That's the kind of scoping that my provides, known as *lexical scoping*, or sometimes as *static* or *compile-time scoping*.

With dynamic scoping, a variable is accessible if it's found in the current scope—or in the scope of any frames (blocks) in its entire subroutine call stack, as determined at runtime. Any functions called have full access to dynamic variables, because they're still globals, just ones with temporary values. Only lexical variables are safe from such tampering.

Declare a function called AUTOLOAD for the package whose undefined function calls you'd like to trap. While running, that package's $AUTOLOAD variable contains the name of the undefined function being called.

```
    sub AUTOLOAD {
        my $color = our $AUTOLOAD;
        $color =~ s/.*::/;
        return "<FONT COLOR='$color'>@_</FONT>";
    }
    #note: sub chartreuse isn't defined.
    print chartreuse("stuff"), "\n";
    # => <FONT COLOR='chartreuse'>stuff</FONT>
```

You want to write a multiway branch statement, much as you can in C using its switch statement or in the shell using case—but Perl seems to support neither.

Use the Switch module, standard as of the v5.8 release of Perl.

```
    use Switch;
    switch ($value) {
```

```
    case 17        { print "number 17"      }
    case "snipe"    { print "a snipe"       }
    case /[a-f]+/i  { print "pattern matched" }
    case [1..10,42] { print "in the list"    }
    case (@array)   { print "in the array"   }
    case (%hash)    { print "in the hash"    }
    else           { print "no case applies" }
}
```

A switch takes an argument and a mandatory block, within which can occur any number of cases. Each of those cases also takes an argument and a mandatory block. The arguments to each case can vary in type, allowing (among many other things) any or all of string, numeric, or regex comparisons against the switch's value. When the case is an array or hash (or reference to the same), the case matches if the switch value corresponds to any of the array elements or hash keys. If no case matches, a trailing else block will be executed.

```
my %traits = (pride => 2, sloth => 3, hope => 14);
switch (%traits) {
    case "impatience"                  { print "Hurry up!\n";      next }
    case ["laziness","sloth"]           { print "Maybe tomorrow!\n"; next }
    case ["hubris","pride"]             { print "Mine's best!\n";    next }
    case ["greed","cupidity","avarice"]   { print "More more more!";   next }
}
# no case applies
# Maybe tomorrow!
# Mine's best!
```

Because each case has a next, it doesn't just do the first one it finds, but goes on for further tests. The next can be conditional, too, allowing for conditional fall through.

Don't recompute sort keys inside a sort.

Doing expensive computations inside the block of a sort is inefficient. By default, the Perl interpreter now uses merge-sorting to implement sort, which means that every sort will call the sort block O(N log N) times. For example, suppose you needed to set up a collection of script files for binary-chop searching.

```
# Sort by SHA512 digest of scripts
# (optimized with the Schwartzian Transform)
@sorted_scripts
 = map  { $_->[0] }             # 3. Extract only scripts
   sort { $a->[1] cmp $b->[1] } # 2. Sort on digests
    map  { [$_, sha512($_)] }    # 1. Precompute digests, store with scripts
        @scripts;
```

This pipelined solution is known as the Schwartzian Transform. Note the special layout, with the three steps lined up under each other. This format is used because it emphasizes the characteristic **map-sort-map** structure of the transform, making it much easier to identify when the technique is being used.

Use reverse to reverse a list.

By default, the sort builtin sorts strings by ascending ASCII sequence. To make it sort by descending sequence instead, you might write:

```
@sorted_results = sort { $b cmp $a } @unsorted_results;
```

But the operation would be much more comprehensible if you wrote:

```
@sorted_results = reverse sort @unsorted_results;
```

That is, if you sorted using the default ordering and then reversed the sorted results afterwards.

Interestingly, in many versions of Perl, it's just as fast (or occasionally even faster) to use an explicitly reversed sort. In recent releases, the reverse sort sequence is recognized and optimized. In older releases, sorting with any explicit block was not optimized, so calling sort without a block is significantly faster, even when the extra cost of the reverse is taken into account.

Another situation in which reversing a list can significantly improve maintainability, without seriously compromising performance, is when you need to iterate "downwards" in a for loop. Instead of writing:

```
for (my $remaining=$MAX; $remaining>=$MIN; $remaining--) {
    print "T minus $remaining, and counting...\n";
    sleep $INTERVAL;
}
```

write:

```
for my $remaining (reverse $MIN..$MAX) {
    print "T minus $remaining, and counting...\n";
    sleep $INTERVAL;
}
```

This approach makes it clear that you intended to count in reverse, as well as making the precise range of $remaining much easier to determine. And, once again, the difference in iteration speed is usually not even noticeable.

Rather than having to puzzle out contexts every time you want to reverse a string, it's much easierand more reliableto develop the habit of always explicitly specifying a **scalar** **reverse** when that's what you want.

```
print scalar reverse("123456789"), "\n"; # => 987654321
```

Use 4-arg substr instead of lvalue substr.

The substr builtin is unusual in that it can be used as an lvalue (i.e., a target of assignment). So you can write things like:

```
substr($addr, $country_pos, $COUNTRY_LEN) = $country_name{$country_code};
```

To avoid those extra steps, in Perl 5.6.1 and later substr also comes in a four-argument model. That is, if you provide a fourth argument to the function, that argument is used as the string with which to replace the substring identified by the first three arguments. So the previous example could be rewritten more efficiently as:

```
substr $addr, $country_pos, $COUNTRY_LEN , country_name{$country_code};
```

Angle brackets are input operators only when they're empty (**<>**), or when they contain a bareword identifier (**<DATA>**), or when they contain a simple scalar variable (**<$input_file>**). If anything else appears inside the angles, they perform shell-based directory look-up instead.

A construct that breaks when you attempt to improve its readability is, by definition, unmaintainable. The file globbing operation has a proper name:

```
my @files = glob($FILE_PATTERN);
```

Use it, and keep the angle brackets strictly for input operations.

Perl's built-in sleep function will only pause your program for an integer number of seconds, even if you give it a floating-point duration:

```
sleep 1.5;        # same as sleep(int(1.5)), so sleeps 1 second
```

the most useful part of this builtin turned out to be its fourth argument, which is supposed to tell select how long to conduct its poll before timing out. It was quickly realized that because this timeout value could be specified in fractions of a second, if select was called with a timeout value but without any streams to poll, like so:

```
select undef, undef, undef, $duration;


sub sleep_for {
    my $duration = shift;
    select undef, undef, undef, $duration;
    return;
}
```

Perl itself encourages the re-use of existing wheels by providing so many built-in functions in the first place. But there are a few gaps in its coverage; a few common tasks that it doesn't provide a convenient builtin to handle.

That's where the Scalar::Util, List::Util, and List::MoreUtils modules can help. They provide commonly needed list and scalar processing functions, which are implemented in C for performance. Scalar::Util and List::Util are part of the Perl standard library (since Perl 5.8), and all three are also available on CPAN.

```
use List::Util qw(first max min sum maxstr minstr shuffle);
use List::MoreUtils qw(all);
my @arr = (1, 2, 3, 4, 5);
my $res = first { $_ > 2 } @arr;
```

```
    print $res, "\n";
    print max(@arr), "\n";
    print min(@arr), "\n";
    print sum(@arr), "\n";

    my @results = all {$_ % 2 == 1} @arr;
    print_array((@arr);

    @arr = qw (aa ab ac ad ae);
    print maxstr(@arr), "\n";
    print minstr(@arr), "\n";

    print_array(shuffle(@arr));
    print_array(shuffle(@arr));
    print_array(shuffle(@arr));
```

Your code will be easier to read and understand if the subroutines always use parentheses and the built-in functions always don't.

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 );
```

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.

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.

```
    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;
    }
```

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.

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

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 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.

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.

```
sub println {
    return print @_, "\n";
}
```

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 argumentssuch as flags or configuration switchesonly a few of which may be needed for any particular invocation.

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.

If default values are needed, set them up first. Separating out any initialization will make your code more readable.

```
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;
}
```

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.

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.

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

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

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

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.

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

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

    return;
}

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

Don't use prototypes.

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;
}

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

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. So always return via an

explicit return.

Note that this rule applies even if your subroutine "doesn't return anything". If the subroutine isn't supposed to return a meaningful value, make it do so explicitly.

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. Single scalar **undef** value becomes a one-element list: **(undef)** in list context.

Exception

As it turns out, though, it's much easier to change how Perl's builtins fail than it is to change how Perl programmers code. You just need to use the standard **Fatal** module:

```
use Fatal qw( open close );

open my $fh, '>', $filename;
print {$fh} $results;
close $fh;
```

The Fatal module is passed a list of builtins and, by the use of dark and terrible magics, it transforms those functions so that they no longer return false on failure; they now throw an exception instead. This means that the last three untested lines of the previous example are now perfectly acceptable. Either each builtin will succeed, or one will fail, at which point that builtin will throw an exception.

Developers who are using your code don't care where your code detected a problem; all they care about is where their code caused the problem. That is, they want to be told the location where the fatal subroutine was called, not the internal location where it actually threw the exception.

And, of course, that's the whole purpose of the standard Carp module: to report exceptions from the caller's point of view. So never use die to throw an exception. The only situation when **die** could reasonably be used instead of **croak( )** is if the error is a purely internal problem within your code, and not the caller's fault in any way. The simple rule of thumb here is that any exception message thrown with a **die** should always start with the words: **'Internal error:...'**.

All the preceding arguments apply to warning messages, too. So always report warning messages using the carp( ) subroutine, instead of the built-in warn.

Unfortunately, most exception messages are written by developers, and for developers (i.e., themselves). Most often, they're written during the testing or debugging process, so they tend to be written in the language of the developers, using the vocabulary of the implementation. So, don't throw exceptions with curt messages that are expressed in the terminology of the implementation. Instead, always **croak** with a detailed message. And phrase it in the vocabulary of the problem space, using concepts that will be familiar to the caller.

Classes, Objects, and Ties

Once associated with a class, something is said to be blessed into that class. There's nothing ecclesiastical or spooky going on here. Blessing merely associates a referent with a class, and this is done with the bless function, which takes one or two arguments. The first is a reference to the thing you want associated with the class; the second is the package with which to make that association.

```
$object = {};                       # hash reference
bless($object, "Data::Encoder");    # bless $object into Data::Encoder class
bless($object);                     # bless $object into current package
```

Once an object has been blessed, calling the ref function on its reference returns the name of its class instead of the fundamental type of referent:

```
$obj = [3, 5];
print ref($obj), " ", $obj->[1], "\n";
bless($obj, "Human::Cannibal");
print ref($obj), " ", $obj->[1], "\n";

ARRAY 5
Human::Cannibal 5
```

As you can see, you can still dereference a reference once it has been blessed. Most frequently, objects are implemented as blessed hash references. You can use any kind of reference you want, but hash references are the most flexible because they allow arbitrarily named data fields in an object.

Although Perl permits it, it's considered poor form for code outside the class to directly access the contents of an object. The point of objects, everyone agrees, is to give you an abstract something with mediated access through designated methods. This lets the maintainer of the class change its implementation without needing to change all application code that uses the class.

The lefthand operand of the **->** operator is said to be the method's invocant. Think of the invocant as the entity on whose behalf the method was called. Methods always involve invocants. Here we have an object method because we invoke the method on an object. We can also have class methods where the invocant is a string representing the package—meaning, of course, the class.

Most classes provide constructor methods, which return new objects. Unlike in some object-oriented languages, constructor methods in Perl are not specially named. In fact, you can name them anything you like. C++ programmers have a penchant for naming their constructors in Perl **new**.

A typical constructor used as a class method looks like this:

```
sub new {
    my $class = shift;
    my $self  = {};        # allocate new hash for object
    bless($self, $class);
    return $self;
}
```

Call the constructor with:

```
$object = Classname->new( );
```

A destructor is a subroutine that runs when an object's referent is garbage collected, which happens when its internal reference count becomes zero. Because it is invoked implicitly by Perl, unlike a constructor, you have no choice in naming a destructor. You must name your destructor method **DESTROY**.

Some languages syntactically allow the compiler to restrict access to a class's methods. Perl does not—it allows code to invoke any method of an object. The author of a class should clearly document the public methods (those that may be used), and the user of a class should avoid undocumented (implicitly private) methods.

Perl doesn't distinguish between methods that can be invoked on a class (class methods) and methods that can be invoked on an object (instance methods). If you want a particular method to be invoked as a class method only, do something like this:

```
use Carp;
sub class_only_method {
    my $class = shift;
    croak "class method invoked on object" if ref $class;
    # more code here
}
```

If you want to allow a particular method to be invoked as an instance method only, do something like this:

```
use Carp;
sub instance_only_method {
    my $self = shift;
    croak "instance method invoked on class" unless ref $self;
    # more code here
}
```

If your code invokes an undefined method on an object, Perl won't complain at compile time, but this will trigger an exception at runtime. Methods are just function calls whose package is determined at runtime. Like all indirect functions, they can have no prototype checking, because that happens at compile time. Even if methods were aware of prototypes, in Perl the compiler

never checks the precise types or ranges of arguments to functions. Perl prototypes are used to coerce a function argument's context, not to check ranges.

You can prevent Perl from triggering an exception for undefined methods by using the AUTOLOAD mechanism to catch calls to nonexistent methods.

Some languages provide special syntax for inheritance. In Perl, each class (package) can put its list of superclasses (parents in the hierarchy) into the package variable @ISA. This list is searched at runtime when a method that is not defined in the object's class is invoked. If the first package listed in @ISA doesn't have the method but that package has its own @ISA, Perl looks first in that package's own @ISA, recursively, before going on.

Here's the canonical object constructor in Perl:

```
sub new {
    my $class = shift;
    my $self  = { };
    bless($self, $class);
    return $self;
}
```

This is the equivalent one-liner:

```
sub new { bless( {}, shift ) }
```

You might also want to separate the memory allocation and blessing step from the instance data initialization step.

```
sub new {
    my $classname  = shift;          # What class are we constructing?
    my $self     = {};               # Allocate new memory
    bless($self, $classname);        # Mark it of the right type
    $self->_init(@_);                # Call _init with remaining args
    return $self;
}

# "private" method to initialize fields.  It always sets START to
# the current time, and AGE to 0.  If invoked with arguments, _init
# interprets them as key+value pairs to initialize the object with.
sub _init {
    my $self = shift;
    $self->{START} = time();
    $self->{AGE}   = 0;
    if (@_) {
        my %extra = @_;
        @$self{keys %extra} = values %extra;
    }
```

```
}
```

Create a method named DESTROY. This will be invoked when there are no more references to the object, or else when the program shuts down, whichever comes first. You don't need to do any memory deallocation here, just any finalization code that makes sense for the class.

```
sub DESTROY {
    my $self = shift;
    printf("$self dying at %s\n", scalar localtime);
}
```

Because of Perl's automatic memory management, an object destructor is rarely needed in Perl. Even when it is, explicit invocation is not only uncalled for, it's downright dangerous. The destructor will be invoked by the run-time system when the object is no longer in use. Most classes don't need a destructor because Perl takes care of simple matters like memory deallocation.

Each data attribute of an object, sometimes named data members or properties, needs its own method for access.

Either write pairs of get and set methods that affect the appropriate key in the object hash, like this:

```
sub get_name {
    my $self = shift;
    return $self->{NAME};
}

sub set_name {
    my $self     = shift;
    $self->{NAME} = shift;
}
```

or make single methods that do both jobs depending on whether they're passed an argument:

```
sub name {
    my $self = shift;
    $self->{NAME} = shift if @_;
    return $self->{NAME};
}
```

When setting a new value, sometimes it may be useful to return not that new value, but the previous one:

```
# returns previously set value if changing it
sub age {
    my $self = shift;
    my $oldage = $self->{AGE};
```

```
        $self->{AGE} = shift if @_;
        return $oldage;
    }
```

By mandating a strictly functional interface, you are free to alter your internal representation later without fear of breaking user code. The functional interface allows you to run arbitrary range checks and take care of any data reformatting or conversion.

Use the standard Class::Struct module's struct to declare data structures reminiscent of those in the C programming language:

```
    use Class::Struct;        # load struct-building module

    struct Person => {          # create a definition for a "Person"
        name   => '$',          # name field is a scalar
        age    => '$',          # age field is also a scalar
        peers  => '@',          # but peers field is an array (reference)
    };

    my $p = Person->new();     # allocate an empty Person struct

    $p->name("Jason Smythe");               # set its name field
    $p->age(13);                          # set its age field
    $p->peers( ["Wilbur", "Ralph", "Fred" ] );  # set its peers field

    # or this way:
    #@{$p->peers} = ("Wilbur", "Ralph", "Fred");

    # fetch various values, including the zeroth friend
    printf "At age %d, %s's first friend is %s.\n", $p->age, $p->name,
$p->peers(0);
```

The Class::Struct::struct function builds struct-like classes on the fly. It creates a class of the name given in the first argument, complete with a constructor named new and per-field accessor methods.

In the structure layout definition, the keys are the names of the fields and the values are the data type. This type can be one of the three base types: '$' for scalars, '@' for arrays, and '%' for hashes.

If all fields are the same type, rather than writing it out this way:

```
    struct Card => {
        name    => '$',
        color   => '$',
        cost    => '$',
        type    => '$',
```

```
    release => '$',
    text    => '$',
};
```

You could use a map to shorten it:

```
struct Card => { map { $_ => '$' } qw(name color cost type release text) };
```

Or, if you're a C programmer who prefers to precede the field name with its type, rather than vice versa, just reverse the order:

```
struct hostent => { reverse qw{
    $ name
    @ aliases
    $ addrtype
    $ length
    @ addr_list
}};
```

The Storable module's **dclone** function will recursively copy (virtually) any data structure. It works on objects, too, correctly giving you back new objects that are appropriately blessed.

```
sub UNIVERSAL::copy {
    my $self = shift;
    unless (ref $self) {
        require Carp;
        Carp::croak("can't copy class $self");
    }
    require Storable;
    my $copy = Storable::dclone($self);
    return $copy;
}
```

Now all objects can be copied, providing they're of the supported types. Classes that provide their own copy methods are unaffected, but any class that doesn't provide its own copy method will pick up this definition. We placed the require on Storable within the function call itself so that you load Storable only if you actually plan to use it. Likewise, we placed the one for Carp inside the test that will end up using it. By using require, we delay loading until the module is actually needed.

Store the method name as a string in a scalar variable and use it where you would use the real method name to the right of the arrow operator:

```
$methname = "flicker";
$obj->$methname(10);        # invokes $ob->flicker(10);
```

```
# invoke three methods on the object, by name
foreach my $m ( qw(start run stop) ) {
    $obj->$m();
}
```

You want to know whether an object is an instance of a particular class or that class's subclasses. Use methods from the special UNIVERSAL class:

```
$obj->isa("HTTP::Message");               # as object method
HTTP::Response->isa("HTTP::Message");       # as class method

if ($obj->can("method_name")) { .... }      # check method validity
```

UNIVERSAL has only a few predefined methods, although you are free to add your own. These are built right into your Perl binary, so they don't take extra time to load. Predefined methods include isa, can, and VERSION. All three may be used for both sorts of invocants: classes and objects.

The isa method reports whether its invocant inherits the class name directly or indirectly from the class name supplied as the argument. This saves having to traverse the hierarchy yourself, and is much better than testing with an exact check against the string returned by the ref built-in. You may even supply a basic type that ref might return as an argument, such as SCALAR, ARRAY, HASH, or GLOB.

The **can** method reports whether its string argument is a valid method for its invocant.

The **VERSION** method checks whether the invocant class has a package global called **$VERSION** that's high enough. Remember, in Perl an all-uppercase function name means that the function will be automatically called by Perl in some way. In this case, it happens when you say:

```
use Some_Module 3.0;
```

If you wanted to add version checking to your Person class explained earlier, add this to *Person.pm*:

```
our $VERSION = "1.01";
```

Imagine you've implemented a class named Person that supplies a constructor named new, and methods such as age and name. Here's the straightforward implementation:

```
package Person;
sub new {
    my $class = shift;
    my $self  = {};
    return bless $self, $class;
}
sub name {
```

```
    my $self = shift;
    $self->{NAME} = shift if @_;
    return $self->{NAME};
}
sub age {
    my $self = shift;
    $self->{AGE} = shift if @_;
    return $self->{AGE};
}
```

Now consider another class, the one named Employee:

```
package Employee;
use Person;
our @ISA = ("Person");
1;
```

```
use Employee;
my $empl = Employee->new( );
$empl->name("Jason");
$empl->age(23);
printf "%s is age %d.\n", $empl->name, $empl->age;
```

Your class's constructor method overrides the constructor of its parent class. You want your constructor to invoke the parent class's constructor.

Learn about the special pseudoclass, SUPER.

```
sub meth {
    my $self = shift;
    $self->SUPER::meth( );
}
```

An overriding constructor should invoke its SUPER's constructor to allocate and bless the object, limiting itself to instantiating any data fields needed.

```
sub new {
    my $classname  = shift;        # What class are we constructing?
    my $self       = $classname->SUPER::new(@_);
    $self->_init(@_);
    return $self;                  # And give it back
}

sub _init {
    my $self = shift;
    $self->{START}   = time();   # init data fields
    $self->{AGE}     = 0;
    $self->{EXTRA}   = { @_ };   # anything extra
```

```
}
```

Carefully use Perl's AUTOLOAD mechanism as something of a proxy method generator so you don't have to create them all yourself each time you want to add a new data field.

```
package Person;
use strict;
use Carp;
our(%ok_field);

# Authorize four attribute fields
for my $attr ( qw(name age peers parent) ) { $ok_field{$attr}++; }

sub AUTOLOAD {
    my $self = shift;
    my $attr = our $AUTOLOAD;
    $attr =~ s/.*::///;
    return unless $attr =~ /[^A-Z]/;  # skip DESTROY and all-cap methods
    croak "invalid attribute method: ->$attr( )" unless $ok_field{$attr};
    $self->{uc $attr} = shift if @_;
    return $self->{uc $attr};
}
1;
```

References and Records

Perl's three built-in types combine with references to produce arbitrarily complex and powerful data structures.

Referents in Perl are typed. This means, for example, that you can't treat a reference to an array as though it were a reference to a hash. Attempting to do so raises a runtime exception. No mechanism for type casting exists in Perl. This is considered a feature.

Explicitly create anonymous arrays and hashes with the **[ ]** and **{ }** composers. This notation allocates a new array or hash, initializes it with any data values listed between the pair of square or curly brackets, and returns a reference to the newly allocated aggregate:

```
$aref = [ 3, 4, 5 ];                          # new anonymous array
$href = { "How" => "Now", "Brown" => "Cow" };      # new anonymous hash
```

Perl also implicitly creates anonymous data types through *autovivification*. This occurs when you indirectly store data through a variable that's currently undefined; that is, you treat that variable as though it holds the reference type appropriate for that operation. When you do so, Perl allocates the needed array or hash and stores its reference in the previously undefined variable.

```
undef $aref;
my @$aref = (1, 2, 3);
print $aref;
ARRAY(0x80c04f0)
```

Syntax for named and anonymous values

Reference to	Named	Anonymous
Scalar	\$scalar	\do{my $anon}
Array	\@array	[LIST]
Hash	\%hash	{ LIST }
Code	\&function	sub { CODE }
Glob	*symbol	open(my $handle, ...); $handle

Although memory allocation in Perl is sometimes explicit and sometimes implicit, memory deallocation is nearly always implicit. You don't routinely have cause to undefine variables. Just let lexical variables (those declared with my) evaporate when their scope terminates; the next time you enter that scope, those variables will be new again. For global variables (those declared with our, fully-qualified by their package name, or imported from a different package) that you want reset, it normally suffices to assign the empty list to an aggregate

variable or a false value to a scalar one.

A record is a single logical unit comprising various different attributes. For instance, a name, an address, and a birthday might compose a record representing a person. C calls such things structs, and Pascal calls them RECORDs. Perl doesn't have a particular name for these because you can implement this notion in different ways. The most common technique in Perl is to treat a `hash` as a record, where the keys of the hash are the record's field names and the values of the hash are those fields' values.

To get the last index number by reference, or the number of items in that referenced array:

```
$last_idx  = $#$aref;
$num_items = @$aref;
```

Or defensively embracing and forcing context:

```
$last_idx  = $#{ $aref };
$num_items = scalar @{ $aref };
```

To check whether something is a array or hash reference:

```
if (ref($someref) ne "ARRAY") {
    die "Expected an array reference, not $someref\n";
}

if (ref($someref) ne "HASH") {
    die "Expected a hash reference, not $someref\n";
}
```

To get a code reference:

```
$cref = \&func;
$cref = sub { ... };
```

To call a code reference:

```
@returned = $cref->(@arguments);
@returned = &$cref(@arguments);
```

If the name of a function is func, you can produce a reference to it by prefixing its name with `\&`. You can also dynamically allocate anonymous functions using the `sub { }` notation. These code references can be stored just like any other reference.

To create a reference to a scalar variable, use the backslash operator:

```
$scalar_ref = \$scalar;      # get reference to named scalar
```

Use ${...} to dereference:

```
print ${ $scalar_ref };      # dereference it
${ $scalar_ref } .= "string"; # alter referent's value
```

You want to print out a data structure. From within your own programs, use the Dumper function from the standard module Data::Dumper.

```
use Data::Dumper;
print Dumper($reference);
```

Or if you'd like output formatted in the same style as the Debugger uses.

```
use Dumpvalue;
Dumpvalue->new->dumpValue($reference);
```

The Data::Dumper module, also included in the standard Perl distribution, has a different approach. It provides a **Dumper** function that takes a list of references and returns a string with a printable (and **eval**able) form of those references.

The Storable module provides a function called dclone that recursively copies its argument:

```
use Storable qw(dclone);
$r2 = dclone($r1);
```

This only works on references or blessed objects of type SCALAR, ARRAY, HASH, or CODE; references of type GLOB, IO, and the more esoteric types are not supported.

Use the Storable module's store and retrieve functions:

```
use Storable;
store(\%hash, "filename");

# later on...
$href = retrieve("filename");        # by ref
%hash = %{ retrieve("filename") };   # direct to hash
```

The store and **retrieve** functions expect binary data using the machine's own byte-ordering. This means files created with these functions cannot be shared across different architectures. **nstore** does the same job **store** does, but keeps data in canonical (network) byte order, at a slight speed cost.

```
{ package BinTree;
  sub insert {
     my($tree, $value) = @_;
     unless ($tree) {
       $tree = {};                    # allocate new node
       $tree->{VALUE}  = $value;
       $tree->{LEFT}   = undef;
       $tree->{RIGHT}  = undef;
```

```
        $_[0] = $tree;                  # $_[0] is reference param!
        return;
    }
    if   ($tree->{VALUE} > $value) { insert($tree->{LEFT},  $value) }
    elsif ($tree->{VALUE} < $value) { insert($tree->{RIGHT}, $value) }
    else                        { warn "dup insert of $value\n"  }
}

sub in_order {
    my($tree) = @_;
    return unless $tree;
    in_order($tree->{LEFT});
    print $tree->{VALUE}, " ";
    in_order($tree->{RIGHT});
}

sub pre_order {
    my($tree) = @_;
    return unless $tree;
    print $tree->{VALUE}, " ";
    pre_order($tree->{LEFT});
    pre_order($tree->{RIGHT});
}

sub post_order {
    my($tree) = @_;
    return unless $tree;
    post_order($tree->{LEFT});
    post_order($tree->{RIGHT});
    print $tree->{VALUE}, " ";
}

sub search {
    my($tree, $value) = @_;
    return unless $tree;
    if ($tree->{VALUE} == $value) {
      return $tree;
    }
    search($tree->{($value < $tree->{VALUE}) ? "LEFT" : "RIGHT"}, $value)
}
}

my $bt;
BinTree::insert($bt, 3);
```

```
BinTree::insert($bt, 5);
BinTree::insert($bt, 1);
BinTree::insert($bt, 7);
BinTree::pre_order($bt);
if (my $res = BinTree::search($bt, 7)) {
 print $res->{VALUE}, "\n";
}
```

Interactivity

Both getopt and getopts can take a second argument, a reference to a hash. If present, option values are stored in $hash{X} instead of $opt_X:

```
use Getopt::Std;

my %option = ();
getopts("Do:", \%option);
print "Debugging mode enabled.\n" if ($option{D});
$option{o} = "-" unless defined $option{o};
print "Writing output to file $option{o}\n" unless $option{o} eq "-";
```

The Getopt::Long module's GetOptions function parses this style of options. It takes a hash whose keys are options and values are references to scalar variables:

```
use Getopt::Long;

my $extract;
my $file;

GetOptions( "extract" => \$extract, "file=s"  => \$file );

if ($extract) {
    print "I'm extracting.\n";
}

die "I wish I had a file" unless defined $file;
print "Working on the file $file\n";
```

Specifier	Value?	Comment
option	No	Given as - -option or not at all
option!	No	May be given as - -option or - -nooption
option=s	Yes	Mandatory string parameter: - -option=somestring
option:s	Yes	Optional string parameter: - -option or - -option=somestring
option=i	Yes	Mandatory integer parameter: - -option=35
option:i	Yes	Optional integer parameter: - -option or - -option=35
option=f	Yes	Mandatory floating point parameter: - -option=3.141
option:f	Yes	Optional floating point parameter: - -option or - -option=3.141

Use the CPAN module Term::ReadKey to put the terminal into cbreak mode, read characters from STDIN, and then put the terminal back into its normal mode:

```
use Term::ReadKey;

sub read_key {
    ReadMode 'cbreak';
    my $key = ReadKey(0);
    ReadMode 'normal';
    return $key;
}
my $key = read_key();
print $key, "\n";
$key = read_key();
print $key, "\n";
```

Naming Conventions

Consistent and coherent code layout is vital, because it determines what the reader of your code sees. But naming conventions are even more important, because they determine how the reader thinks about your program.

Well-chosen identifier names convey to the reader the meaning of the data in variables, the behaviour and results of subroutines, and the features and purpose of classes and other data types. They can help to make the data structures and algorithms used in a program explicit and unambiguous. They can also function as a reliable form of documentation, and as a powerful debugging aid.

Syntactic consistency means that all identifiers should conform to a predictable and recognizable grammatical structure. In other words, if one variable name has an adjective_noun structure, all variable names should be adjective_noun.

Semantic consistency means that the names you choose should clearly and accurately reflect the purpose, usage, and significance of whatever you're naming. In other words, a name like **@data** is a poor choice (compared to, say, **@sales_records**) because it fails to tell the reader anything important about the contents of the array or their significance in your program.

```
my $next_client;
# not: $next_elem

my $prev_appointment;
# not: $prev_elem

my $estimated_nett_worth;
# not: $value

my $next_node;
# not: $node

my $root_node;
# not: $root

my $final_total;
# not: $sum

my $cumulative_total;
# not: $partial_sum
```

```
my $running_total = 0;
my $games_count   = 0;

while (my $next_score = get_next_score_for($curr_player)) {
    $running_total += $next_score;
    $games_count++;
}
print "After $games_count: $running_total \n";
```

Note that the rules for creating variables (**$adjective_noun**) are the opposite of those for naming classes and packages (**Noun::Adjective**). This is deliberate, and designed to help readers distinguish between the two types of names. The more conventional grammatical structure (adjective before the noun) is used for the more frequently used type of name (i.e., for variables), which improves the overall readability of the code. At the same time, namespace names stand out better because of their unusual reversed syntax.

Adding a preposition to the end of the name makes hash and array accesses much more readable:

```
my %title_of;
my %ISBN_for;
my @sales_from;
```

For subroutines and methods, a suitable grammatical rule for forming names is:

```
sub get_record;                    # imperative_verb noun
sub get_record_for;                # imperative_verb noun preposition

sub eat_cookie;                    # imperative_verb noun
sub eat_previous_cookie;           # imperative_verb adjective noun

sub build_profile;                  # imperative_verb noun
sub build_execution_profile;        # imperative_verb adjective noun

sub build_execution_profile_using;
# imperative_verb adjective noun participle
```

A special case can be made for subroutines that return boolean values, and for variables that store them. These should be named for the properties or predicates they test, in such a way that the resulting conditional expressions read naturally. Often that rule will mean they begin with **is_** or **has_**, but not always.

```
sub is_valid;
sub metadata_available_for;
```

```
sub has_end_tag;
my $loading_finished;
my $has_found_bad_record;
```

You can minimize the chances of making this mistake in the first place by always appending the suffix **_ref** to any variable that is supposed to store a reference.

```
sub pad_str {
    my ($text, $opts_ref) = @_;

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

    return $SPACE x $left . $text . $SPACE x $right;
}
```

Because hash entries are typically accessed individually, it makes sense for the hash itself to be named in the singular. That convention causes the individual accesses to read more naturally in the code. Moreover, because hashes often store a property that's related to their key, it's often even more readable to name a hash with a singular noun followed by a preposition.

```
my %option;
my %title_of;
my %count_for;
my %is_available;
```

On the other hand, array values are more often processed collectively, in loops or in **map** or **grep** operations. So it makes sense to name them in the plural, after the group of items they store.

```
my @events;
my @handlers;
my @unknowns;
```

```
for my $event (@events) {
    push @unknowns, grep { ! $_->handle($event) } @handlers;
}
print map { $_->err_msg } @unknowns;
```

Since neither spaces nor hyphens are valid characters in Perl identifiers, use the next closest available alternative: the underscore. Underscores correspond better to the default natural-language word separator (a space) because they impose a visual gap between the words in an identifier.

```
FORM:
for my $tax_form (@tax_form_sequence) {
     my $notional_tax_paid
        = $tax_form->{reported_income} * $tax_form->{effective_tax_rate};

     next FORM if $notional_tax_paid  < $MIN_ASSESSABLE;

     $total_paid
        += $notional_tax_paid - $tax_form->{allowed_deductions};
}
```

To help make it clear what kind of referent an identifier is naming:

- Use lowercase only for the names of subroutines, methods, variables, and labeled arguments ($controller, new( ), src=>$fh).
- Use mixed-case for package and class names (IO::Controller).
- Use uppercase for constants ($SRC, $NODE)

Note that the only exception to this guideline should be in identifiers that include a proper name, a standard abbreviation, or a unit of measurement.

If you choose to abbreviate an identifier, abbreviate it by retaining the start of each word. This generally produces much more readable names than other approaches.

```
my $len = length $desc;
my $ctrl_char = '\N{ESCAPE}';
```

The standard single letter iterator variables $i, $j, $k, $n, $x, $y, $z are often acceptable in nested loops, especially when the indices are coordinates of some kind.

```
sub swap_domain_and_range_of {
     my ($table_ref) = @_;

     my @pivotted_table;
     for my $x (0..$#{$table_ref}) {
         for my $y (0..$#{$table_ref->[$x]}) {
             $pivotted_table[$y][$x] = $table_ref->[$x][$y];
         }
     }

     return \@pivotted_table;
}
```

A utility subroutine exists only to simplify the implementation of a module or class. It is never supposed to be exported from its module, nor ever to be used in client code.

Always use an underscore as the first "letter" of any utility subroutine's name. A leading underscore is ugly and unusual and reserved (by ancient C/Unix convention) for non-public components of a system. The presence of a leading underscore in a subroutine call makes it immediately obvious when part of the implementation has been mistaken for part of the interface.

```
# Cache of previous results, minimally initialized...
my @fib_for = (1,1);

# Extend cache when needed...
sub _find_fib {
    my ($n) = @_;

    # Walk up cache from last known value, applying Fn = Fn-1 + Fn-2...
    for my $i (@fib_for..$n) {
        $fib_for[$i] = $fib_for[$i-1] + $fib_for[$i-2];
    }

    return;
}

# Return Fibonacci number N
sub fib {
    my ($n) = @_;

    # Verify argument in computable range...
    croak "Can't compute fib($n)" if $n < 0;

    # Extend cache if necessary...
    if ( !defined $fib_for[$n] ) {
        _find_fib($n);
    }

    # Look up value in cache...
    return $fib_for[$n];
}
```

If you're creating a literal character string and you definitely intend to interpolate one or more variables into it, use a double-quoted string:

```
my $spam_name = "$title $first_name $surname";
my $pay_rate  = "$minimal for maximal work";
```

If you're creating a literal character string and not intending to interpolate any variables into it, use a single-quoted string:

```
my $spam_name = 'Dr Lawrence Mwalle';
my $pay_rate  = '$minimal for maximal work';
```

If your uninterpolated string includes a literal single quote, use the q{...} form instead:

```
my $spam_name = q{Dr Lawrence ('Larry') Mwalle};
my $pay_rate  = q{'$minimal' for maximal work};
```

If your uninterpolated string includes both a literal single quote and an unbalanced brace, use square brackets as delimiters instead:

```
my $spam_name = q[Dr Lawrence }Larry{ Mwalle];
my $pay_rate  = q['$minimal' for warrior's work {{:-)];
```

Reserving interpolating quoters for strings that actually do interpolate something can help you avoid unintentional interpolations, because the presence of a $ or @ in a single-quoted string then becomes a sign that something might be amiss. Likewise, once you become used to seeing double quotes only on interpolated strings, the absence of any variable in a double-quoted string becomes a warning sign. So these rules also help highlight missing intentional interpolations.

For sequences of "parallel" strings, choose the most general delimiters required and use them consistently throughout the set:

```
my $title           =  q[Perl Best Practices];
my $publisher       =  q[O'Reilly];
my $end_of_block    =  q[}];
my $closing_delim   =  q['}];
my $citation        = qq[$title ($publisher)];
```

Don't use **""** or **''** for an empty string.

```
$error_msg = q{};
# Empty string
```

The **Readonly** CPAN module exports a single subroutine (**Readonly( )**) that expects two arguments: a scalar, array, or hash variable, and a value. The value is assigned to the variable, and then the variable's "read-only" flag is set, to prevent any further assignments. Note the use of all-uppercase in the variable name.

```
use Readonly;
Readonly my $MOLYBDENUM_ATOMIC_NUMBER => 42;

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

```
Readonly my $DOUBLE_QUOTE   => q{"};
Readonly my $COMMA          => q{,};

Readonly my %PERMISSIONS_FOR => (
    USER_ONLY       => oct(600),
    NORMAL_ACCESS   => oct(644),
    ALL_ACCESS      => oct(666),
);
```

So Perl provides a convenient mechanism for making large numbers easier to read: you can use underscores to "separate your thousands".

```
$US_GDP                = 10_990_000_000_000;
$US_govt_revenue       =  1_782_000_000_000;
$US_govt_expenditure   =  2_156_000_000_000;
```

If a string has embedded newline characters, but the entire string won't fit on a single source line, then break the string after each newline and concatenate the pieces:

```
$usage = "Usage: $0 <file> [-full]\n"
         . "(Use -full option for full dump)\n"
         ;
```

Single-quoting the marker forces the heredoc to not interpolate variables. That is, it acts just like a single-quoted string:

```
Readonly my $GRIPE => <<'END_GRIPE';
$minimal for maximal work
END_GRIPE

print $GRIPE;
# Prints: $minimal for maximal work
```

Double-quoting the marker ensures that the heredoc string is interpolated, just like a double-quoted string:

```
Readonly my $GRIPE => <<"END_GRIPE";
$minimal for maximal work
END_GRIPE

print $GRIPE;
# Prints: 4.99 an hour for maximal work
```

Stick to using only lexical variables (my), unless you genuinely need the functionality that only a package or punctuation variable can provide.

Alternatives to built-in variables

Variable	Purpose	Alternative
$1, **$2**, **$3**, etc.	Store substrings captured from the previous regex match	Assign captures directly using list context regex matching, or unpack them into lexical variables immediately after the match (see Chapter 12). Note that these variables are still acceptable in the replacement string of a substitution, because there is no alternative. For example: s{($DD)/($MMM)/($YYYY)}{$3-$2-$1}xms
$&	Stores the complete substring most recently matched by a regex	Place an extra set of capturing parentheses around the entire regex, or use **Regexp::MatchContext** (see the "Match Variables" guideline later in this chapter).
$'	Stores the substring that preceded the most recent successful regex match	Place a **((?s).*?)** at the beginning of the regex to capture everything up to the start of the pattern you are actually interested in, or use **Regexp::MatchContext**.
$'	Stores the substring that followed the most recent successful regex match	Place a **((?s).*)** at the end of the regex to capture everything after the pattern you are actually interested in, or use **Regexp::MatchContext**.
$*	Controls newline matching in regexes	Use the **/m** regex modifier.
$.	Stores the current line number of the current input stream	Use **$fh->input_line_number( )**.
$|	Controls autoflushing of the current output stream	Use **$fh->autoflush( )**.
$"	Array element separator when interpolating into	Use an explicit **join**.

Alternatives to built-in variables

Variable	Purpose	Alternative
	strings	
$%, $=, $-, $~, $^, $:, $^L, $^A	Control various features of Perl's **format** mechanism	Use **Perl6::Form::form** instead (see Chapter 19).
$[	Determines the starting index of arrays and strings	Never change the starting index from zero.
@F	Stores the result of autosplitting the current line	Don't use the **-a** command-line flag when invoking *perl*.
$^W	Controls warnings	Under Perl 5.6.1 and later, specify **use warnings** instead.

By using a local declaration when making that change, you restrict its effects to the dynamic scope of the declaration:

```
use YAML;
local $YAML::Indent = 4;
# Indent is 4 until control exits current scope
```

That is, by prefacing the assignment with the word local, you can temporarily replace the package variable $YAML::Indent until control reaches the end of the current scope. So any calls to the various subroutines in the YAML package from within the current scope will see an indent value of 4. And after the scope is exited, the previous indent value (whatever it was) will be restored.

Many people seem to think that a localized variable keeps its pre-localization value. It doesn't. Whenever a variable is localized, its value is reset to **undef.**

To correctly localize a package variable but still retain its pre-localization value, you need to write this instead:

```
Readonly my $DEFAULT_INDENT => 4;

# and later...

use YAML;
```

```
    local $YAML::Indent = $DEFAULT_INDENT;
```

Using **local** is the cleanest and most robust way to temporarily change the value of a global variable. It should always be applied in the smallest possible scope, so as to minimize the effects of any "ambient behaviour" the variable might control.

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

if (@ARGV) {
    local $INPUT_RECORD_SEPARATOR  = undef;   # Slurp mode
    local $OUTPUT_RECORD_SEPARATOR = $SPACE;  # Autoappend a space to every print
    local $OUTPUT_AUTOFLUSH        = 1;       # Flush buffer after every print



    # Slurp, mutilate, and spindle...
    $text = <>;
    $text =~ s/\n/[EOL]/gxms;
    print $text;
}
```

One particularly easy way to introduce subtle bugs is to forget that **$_** is often an alias for some other variable. Any assignment to **$_** or any other form of transformation on it, such as a substitution or transliteration, is probably changing some other variable. So any change applied to **$_** needs to be scrutinized particularly carefully.

The preceding assignments are much better written as:

```
    # Replace broken frames...

    $frames[-1] = $active{top};
    # 1st-last frame (i.e., final frame)

    $frames[-2] = $active{prev};
    # 2nd-last frame

    $frames[-3] = $active{backup};
    # 3rd-last frame
```

Using negative indices is good practice, because the leading minus sign makes the index stand out as unusual, forcing the reader to think about what that index means and marking any "from the end" indices with an obvious prefix.

The previous examples would be even less cluttered (and hence more readable) using an array slice and a hash slice:

```
    @frames[-1,-2,-3]
```

```
      = @active{'top', 'prev', 'backup'};
```

```
@frames[-1,-2,-3]
```

is exactly the same as:

```
($frames[-1], $frames[-2], $frames[-3])
```

```
@active{'top', 'prev', 'backup'}
```

is exactly the same as:

```
($active{'top'}, $active{'prev'}, $active{'backup'})
```

A slice-to-slice assignment like:

```
@frames[-1,-2,-3]
    = @active{'top', 'prev', 'backup'};
```

can also be written as:

```
 @frames[  -1,    -2,     -3    ]
= @active{'top', 'prev', 'backup'};
```

A more readable and more scalable approach in such cases is to factor out the index/key equivalences in a separate tabular data structure:

```
Readonly my %CORRESPONDING => (
   # Key of        Index of
   # %active...    @frames...
      'top'         =>  -1,
      'prev'        =>  -2,
      'backup'      =>  -3,
      'emergency'   =>  -4,
      'spare'       =>  -5,
      'rainy day'   =>  -6,
      'alternate'   =>  -7,
      'default'     =>  -8,
   );

   @frames[ values %CORRESPONDING ] = @active{ keys %CORRESPONDING };
```

Hence it's better to place the **next**, **last**, **redo**, **return**, **goto**, **die**, **croak**, and **throw** keywords in the most prominent position on their code line. In other words, they should appear as far to the left as possible.

```
sub find_anomolous_sample_in {
      my ($samples_ref) = @_;

      MEASUREMENT:
      for my $measurement (@{$samples_ref}) {
```

```
            last MEASUREMENT if $measurement < 0;

            my $floor = int($measurement);
            next MEASUREMENT if $floor == $measurement;

            my $allowed_inaccuracy = scale($EPSILON, $floor);
            return $measurement
                if $measurement-$floor > $allowed_inaccuracy;
        }
        return;
    }
```

```
RESULT:
for my $n (4..$MAX) {
    next RESULT if odd($n);
    print $result[$n];
}
```

The advantage with this version is that subsequent readers of the code no longer have to work out the logic of the loop. The code itself says explicitly:

"n from 4 to MAX, skipping values that are odd."

in Perl 5.6 and later, the **values** function returns a list of aliases to the actual values of the hash, rather than just a list of copies. So if you change the iterator variable (for example, assigning **'[DELETED]'** to **$translated_word**), you're actually changing the corresponding original value inside the hash.

Aliasing is the process of putting a second (or third, or nth) "Hi-my-name-is-also..." sticker on a single box. **Perl subroutines do this all the time.** For example, if you call get_passwd($user), then inside the call to get_passwd( ) the name $_[0] is temporarily attached to the container whose original name is $user. That container now has two names: one that's used inside the subroutine and one that's used outside.

Anything you do to an alias (e.g., get its value, increment it, print it, assign a new value to it) is really being done to the original variablebecause there's really only one variable, no matter how many separate names you give it.

The alternative is to use Perl's built-in map function. This function is specifically aimed at those situations when you want to process a list of values, to create some kind of related list. For example, to produce a list of square roots from a list of numbers:

```
my @sqrt_results = map { sqrt $_ } @results;
```

There are a couple of other advantages that aren't quite as obvious. For example, when you use map, most of your looping and list generation is being done in heavily optimized compiled C code, not in interpreted Perl. So it's usually being done considerably faster.

In addition, the map knows in advance exactly how many elements it will eventually process, so it can preallocate sufficient space in the list it's returning. Or rather it can usually preallocate sufficient space. If the map's block returns more than one value for each element of the original list, then extra allocations will still be necessary. But, even then, not as many as the equivalent series of push statements would require.

Finally, on a more abstract level, a map is almost always used to transform a sequence of data, so seeing a map immediately suggests to the reader that a data transformation is intended. And the syntax of the function makes it easy to visually locate both the transformation itself (what's in the braces) and the data it's being applied to (what's after the braces).

```
# Identify candidates who are unfit for the cut-and-thrust of politics...

my @disqualified_candidates
    = grep {cannot_tell_a_lie($_)} @candidates;
```

Using the first function often results in code that is both more comprehensible and more efficient:

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

# Find a juicy story...
my $scapegoat
   = first { chopped_down_cherry_tree($_) }  @disqualified_candidates;
```

There is, however, a particular case where map and grep are not better than an explicit for loop: when you're transforming an array *in situ*. In other words, when you have an array of elements or a list of lvalues and you want to replace each of them with a transformed version of the original.

But the map statement has to allocate extra memory to store the transformed values and then assign that temporary list back to the original array. That process could become expensive if the list is large or the transformation is repeated many times.

In contrast, the equivalent for block can simply reuse the existing memory in the array:

```
for my $measurement (@temperature_measurements) {
   $measurement = F_to_K($measurement);
}
```

One particular feature of the way the **map**, **grep**, and **first** functions work can easily become a source of subtle errors. These functions all use the **$_** variable to pass each list element into their associated block. But, for better efficiency, these functions alias **$_** to each list value they're iterating, rather than copying each value into **$_**. If the block you give to a **map**, **grep**, or **first** modifies **$_** in any way, then it's actually modifying an alias to some element of the function's list.

Indeed, you could build a simple tabular structure to determine the correct salutation, by cascading ternaries instead of ifs, like so:

```
        # Name format...                                  # Salutation...
my $salute = $name eq $EMPTY_STR                        ? 'Dear Customer'
       : $name =~ m/ \A((?:Sir|Dame) \s+ \S+) /xms      ? "Dear $1"
       : $name =~ m/ (.*), \s+ Ph[.]?D \z    /xms       ? "Dear Dr $1"
       :                                                  "Dear $name"
       ;
```

The efficiency of this series of tests will be exactly the same as the preceding cascaded-**if** version, so there's no advantage in that respect. The advantages of this approach are in terms of readability and comprehensibility. A second advantage of the ternary version is that (if you squint a little) it looks like a table. The ternary version is also considerably more compact, and requires two-thirds fewer lines than the equivalent cascaded **if**. That makes it far easier to keep the code on one screen as additional alternatives are added.

The final advantage of using ternaries instead of an **if** cascade is that the syntax of the ternary operator is much stricter. In a regular cascaded **if** statement, it's easy to accidentally leave off the final unconditional **else**. However, there is no way to make the same mistake using a ternary cascade. If you do, Perl will immediately (and lethally) inform you that leaving out the final alternative is a **syntax error**.

Checking data as soon as it's available means that you can short-circuit sooner if the data is unacceptable.

```
CLIENT:
for my $client (@clients) {
    # Verify active client...
    next CLIENT if !$client->{activity};

    # Compute current client value and verify client is worth watching...
    my $value = $client->{volume} * $client->{rate};
    next CLIENT if $value < $WATCH_LEVEL;

    # Compute likely client future value and verify client is worth keeping...
    my $projected = $client->{activity}* $value;
```

```
        next CLIENT if $projected < $KEEP_LEVEL;

        # Add in client's expected contribution...
        $total += $projected * $client->{volatility};
    }
```

Don't use unnecessary parentheses for builtins and "honorary" builtins. Perl's many built-in functions are effectively keywords of the language, so they can legitimately be called without parentheses, except where it's necessary to enforce precedence.

Separate complex keys or indices from their surrounding brackets.

```
    $candidates[$i] = $incumbent{ $candidates[$i]{ get_region() } };

    print $incumbent{ $largest_gerrymandered_constituency };
```

Place a comma after every value in a multiline list. Adding an extra trailing comma (which is perfectly legal in any Perl list) also makes it much easier to reorder the elements of the list.

```
    my @dwarves = (
        'Happy',
        'Sleepy',
        'Dopey',
        'Sneezy',
        'Grumpy',
        'Bashful',
        'Doc',
    );
```

Setting your right margin at 78 columns maximizes the usable width of each code line whilst ensuring that those lines appear consistently on the vast majority of display devices.

In *vi*, you can set your right margin appropriately by adding:

```
    set textwidth=78
```

Tabs are a bad choice for indenting code, even if you set your editor's tabspacing to four columns. Tabs do not appear the same when printed on different output devices, or pasted into a word-processor document, or even just viewed in someone else's differently tabspaced editor. So don't use tabs alone or (worse still) intermix tabs with **spaces**.

For example, if you use *vim*, you can include the following directives in your *.vimrc* file:

```
    set tabstop=4
```

```
    "An indentation level every four columns"

    set expandtab
    "Convert all tabs typed into spaces"

    set shiftwidth=4
    "Indent/outdent by four columns"

    set shiftround
    "Always indent/outdent to the nearest tabstop"
```

Never place two statements on the same line. Note that this guideline applies even to map and grep blocks that contain more than one statement. You should write:

```
    my @clean_words
       = map {
            my $word = $_;
            $word =~ s/$EXPLETIVE/[DELETED]/gxms;
            $word;
         } @raw_words;
```

Break each piece of code into sequences that achieve a single task, placing a single empty line between each sequence. To further improve the maintainability of the code, place a one-line comment at the start of each such paragraph, describing what the sequence of statements does.

Paragraphs are one way of aggregating small amounts of related information, so that the resulting "chunk" can fit into a single slot of the reader's limited short-term memory.

Adding comments at the start of each paragraph further enhances the chunking by explicitly summarizing the purpose of each chunk.

Don't cuddle an else.

A "cuddled" else looks like this:

```
    } else {
```

An uncuddled else looks like this:

```
    }
    else {
```

Tables are another familiar means of chunking related information, and of using physical layout to indicate logical relationships. When setting out code, it's

often useful to align data in a table-like series of columns. Consistent indentation can suggest equivalences in structure, usage, or purpose.

```
my @months = qw(
    January  February    March
    April    May         June
    July     August      September
    October  November    December
);

my %expansion_of = (
    q{it's}      => q{it is},
    q{we're}     => q{we are},
    q{didn't}    => q{did not},
    q{must've}   => q{must have},
    q{I'll}      => q{I will},
);
```

Take a similar tabular approach with sequences of assignments to related variables, by aligning the assignment operators:

```
$name   = standardize_name($name);
$age    = time - $birth_date;
$status = 'active';
```

Alignment is even more important when assigning to a hash entry or an array element. In such cases, the keys (or indices) should be aligned in a column, with the surrounding braces (or square brackets) also aligned.

```
$ident{ name    } = standardize_name($name);
$ident{ age     } = time - $birth_date;
$ident{ status  } = 'active';
```

A cleaner solution is to break long lines before an operator. That approach ensures that each line of the continued expression will start with an operator, which is unusual in Perl code.

When a broken expression is continued over multiple lines, it is good practice to place the terminating semicolon on a separate line, indented to the same column as the start of the continued expression.

As the reader's eye scans down through the leading operators on each line, encountering a semicolon instead makes it very clear that the continued expression is now complete.

```
push @steps, $steps[-1]
           + $radial_velocity * $elapsed_time
           + $orbital_velocity * ($phase + $phase_shift)
           - $DRAG_COEFF * $altitude
```

```
            ;
```

Breaking at operators of higher precedence encourages the unwary reader to misunderstand the computation that the expression performs.

```
push @steps, $steps[-1]
           + $radial_velocity * $elapsed_time
           + $orbital_velocity
               * ($phase + $phase_shift)
           - $DRAG_COEFF * $altitude
           ;
```

Often, the long statement that needs to be broken will be an assignment. A better approach when breaking assignment statements is to break before the assignment operator itself, leaving only the variable being assigned to on the first line.

```
$predicted_val
   = ($minimum + $maximum) / 2
     + $predicted_change * max($fudge_factor, $local_epsilon);
```

In other words, break a series of ternary operators before every colon, aligning the colons with the operator preceding the first conditional. Doing so will cause the conditional tests to form a column. Then align the question marks of the ternaries so that the various possible results of the ternary also form a column. Finally, indent the last result (which has no preceding question mark) so that it too lines up in the results column.

This special layout converts the typical impenetrably obscure ternary sequence into a simple **look-up** table: for a given condition in column one, use the corresponding result from column two.

```
            # When their name is...                    Address them as...
my $salute = $name eq $EMPTY_STR                       ? 'Customer'
           : $name =~ m/\A((?:Sir|Dame) \s+ \S+) /xms   ? $1
           : $name =~ m/(.*), \s+ Ph[.]?D \z    /xms    ? "Dr $1"
           :                                            $name
           ;

my $name = defined $customer{name} ? $customer{name}
         :                           'Sir or Madam'
         ;
```

Miscellanea

Maintaining control over the creation and modification of your source code is utterly essential for robust team-based development.

Revision control systems such as *RCS*, *CVS*, ***Subversion***, *Monotone*, *darcs*, ***Perforce***, *GNU arch*, or *BitKeeper* can protect against calamities, and ensure that you always have a working fallback position if maintenance goes horribly wrong. The various systems have different strengths and limitations, many of which stem from fundamentally different views on what exactly revision control is. So it's a good idea to audition the various revision control systems and find the one that works best for you.

Pragmatic Version Control Using Subversion, by Mike Mason (Pragmatic Bookshelf, 2005) and Essential CVS, by Jennifer Vesperman (O'Reilly, 2003) are useful starting points.

The only way to know for sure which of twoor morealternatives will perform better is to actually time each of them. The standard **Benchmark** module makes that easy.

```
my @data = (1, 1, 3, 5, 5, 7, 7, 9);

sub compare {
    my ($title) = @_;

    print "\n[$title]\n";

    # Create a comparison table of the various timings, making sure that
    # each test runs at least 10 CPU seconds...
    use Benchmark qw( cmpthese );
    cmpthese -10, {
        anon   => 'my @uniq = unique_via_anon(@data)',
        grep   => 'my @uniq = unique_via_grep(@data)',
        slice  => 'my @uniq = unique_via_slice(@data)',
    };

    return;
}

compare('8 items, 10% repetition');

[8 items, 10% repetition]
          Rate  anon slice  grep
anon  217165/s    --  -43%  -48%
slice 380007/s   75%    --   -9%
grep  419676/s   93%   10%    --
```

The **cmpthese**( ) subroutine is passed a number, followed by a reference to a hash of tests. The number specifies either the exact number of times to run each test (if the number is positive) or the absolute number of CPU seconds to run the test for (if the number is negative). Typical values are around 10,000 repetitions or 10 CPU seconds, but the module will warn you if the test is too short to produce an accurate benchmark.

Whenever you decide to add caching to a computation, it's essential to benchmark the resulting code, to make sure that the cache look-up costs aren't more expensive that the computation itself.

```
use Benchmark qw( cmpthese );

my @sqrt_of = map { sqrt $_ } 0..255;
cmpthese -10, {
   recompute   => q{ for my $n (0..255) {my $res = sqrt $n }},
   look_up     => q{ for my $n (0..255) {my $res = $sqrt_of[$n] }},
};
#             Rate   look_up   recompute
#look_up     5726/s       --         -1%
#recompute   5787/s       1%         --
```

How do you find those places where optimization will do the most good? By understanding where your application spends most of its time. And the easiest way to do that is to *profile* your program using the standard **Devel::DProf** module, which can determine how long your application spends within each subroutine in your source code.

```
perl -d:DProf application.pl datafile
```

The -d: debugging flag is a shorthand for -MDevel::, so specifying -d:DProf is the same as specifying -MDevel::DProf. That tells *perl* to include that profiling module before the start of the *application.pl* source.

The module itself simply watches every subroutine call within your code, noting how much time elapses between the invocation and return, and adding that duration to a record of the total time spent in each subroutine. At the end of the program, the module creates a file called *tmon.out* in the current directory.

It's possible to directly interpret the raw data in the file (see the module docs for details), but much easier to understand it by passing it through the standard *dprofpp* application.

```
dprofpp tmon.out
```

```
Total Elapsed Time = 0.558183 Seconds
  User+System Time = 0.507183 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
```

```
 23.6   0.120  0.398    19   0.0063 0.0210  main::BEGIN
 13.8   0.070  0.070     2   0.0350 0.0350  charnames::charnames
 9.86   0.050  0.050     7   0.0071 0.0071  DynaLoader::dl_load_file
 7.89   0.040  0.169     6   0.0066 0.0282  CookBook::Utils::BEGIN
 5.92   0.030  0.040     6   0.0050 0.0066  ActiveState::Path::BEGIN
 3.94   0.020  0.020     4   0.0050 0.0050  Exporter::as_heavy
 3.94   0.020  0.020    10   0.0020 0.0020  vars::import
 3.94   0.020  0.060     3   0.0067 0.0199  ActivePerl::Config::BEGIN
 3.94   0.020  0.040     6   0.0033 0.0066  Time::localtime::BEGIN
 3.94   0.020  0.030     7   0.0028 0.0043  charnames::BEGIN
 1.97   0.010  0.010     1   0.0100 0.0100  warnings::BEGIN
 1.97   0.010  0.010     3   0.0033 0.0033  AutoLoader::import
 1.97   0.010  0.010     3   0.0033 0.0033  BinTree::search
 1.97   0.010  0.010     3   0.0033 0.0033  vars::BEGIN
 1.97   0.010  0.020     3   0.0033 0.0066  Time::tm::BEGIN
```

If you need finer-grained profiling, the Devel::SmallProf CPAN module allows you to count how many times each line of your program is executed, which makes it easier to determine precisely what statement is causing a particular subroutine to be expensive:

```
perl -d:SmallProf application.pl datafile
```

The result is a file named smallprof.out, which is actually a copy of your source code with each line prefixed by the number of times it was executed, the total amount of time spent executing the line, and the line number. Although the module lacks a utility like *dprofpp* to summarize its output, it is an excellent investigative tool once you have identified the main suspects using Devel::DProf.

Indirect filehandles provide a much cleaner and less error-prone alternative to bareword filehandles, and from Perl 5.6 onwards they're as easy to use as barewords. Whenever you call open with an undefined scalar variable as its first argument, open creates an anonymous filehandle (i.e., one that isn't stored in any symbol table), opens it, and puts a reference to it in the scalar variable you passed.

So you can open a file and store the resulting filehandle in a lexical variable, all in one statement, like so:

```
    open my $FILE, '<', $filename
       or croak "Can't open '$filename': $OS_ERROR";
```

Use while (<>), not for (<>).

Whatever the reason, using a for loop to iterate input is a very inefficient and brittle solution. The iteration list of a for loop is (obviously) a list context. So in the example, the <> operator is called in a list context. Evaluating <> in list context causes it to read in every line it can, building a temporary list as it does.

Once the input is complete, that list becomes the list to be iterated by the for.

In contrast, an equivalent while loop:

```
while (my $line = <>) {
    $line =~ s/$EXPLETIVE/[DELETED]/gxms;
    print $line;
}
```

reads and processes only one line at a time. This version can be used interactively, and never allocates more memory than is needed to accommodate the longest individual line. So use a while instead of a for when reading input.

Whenever you do need to read in an entire file at once, the syntax shown in the final example of the previous guideline is the right way to do it:

```
my $code = do { local $/; <$in> };
```

Always put filehandles in braces within any **print** statement.

```
print {$file} $name, $rank, $serial_num, "\n";
print {*STDERR} $name, $rank, $serial_num, "\n";
```

Another acceptable alternative is to load the IO::Handle module and then use Perl's object-oriented I/O interface:

```
use IO::Handle;
$file->print( $name, $rank, $serial_num, "\n" );
*STDERR->print( $name, $rank, $serial_num, "\n" );
```

Because programs so often need to prompt for interactive input and then read that input, it's probably not surprising that there would be a CPAN module to make that process easier. It's called IO::Prompt and it exports only a single subroutine: prompt( ). At its simplest, you can just write:

```
use IO::Prompt;
my $line = prompt 'Enter a line: ';
```

The prompt() subroutine can also control the echoing of characters.

```
my $password = prompt 'Password: ', -echo => '*';
```

You can even prevent echoing entirely (by echoing an empty string in place of each character):

```
my $password = prompt 'Password: ', -echo => $EMPTY_STR;
```

prompt( ) can return a single key-press (without requiring the Return key to be pressed as well):

```
my $choice = prompt 'Enter your choice [a-e]: ', -onechar;
```

It can ignore inputs that are not acceptable:

```
my $choice = prompt 'Enter your choice [a-e]: ', -onechar,
             -require => { 'Must be a, b, c, d, or e: ' => qr/[a-e]/xms };
```

Recommended Core Modules

Module name	Description	Since
base	Specifies the base classes of the current package at compile time	5.005
Benchmark	Provides utilities to time fragments of Perl code	5.003
Carp	Provides subroutines that warn or throw exceptions, reporting the problem from the caller's location	5.6
charnames	Enables the use of character names via **\N{CHARNAME}** string literal escapes	5.6
Data::Dumper	Converts data structures into string representations of Perl code	5.005
Devel::DProf	Profiles Perl code	5.6
English	Defines readable English names for special variables	5.003
Fatal	Replaces functions and subroutines with equivalents that either succeed or throw an exception	5.005
File::Glob	Implements command-line filename globbing	5.6
File::Temp	Provides a safe and efficient way to create temporary files	5.6
Getopt::Long	Parses command-line options	5.003
IO::File	Creates I/O objects connected to files	5.004
IO::Handle	Acts as the base class for filehandles and objects	5.004
List::Util	Provides additional list-processing utilities missing from the core language	5.8
Memoize	Optimizes subroutines by caching their return values and reusing them	5.003
overload	Allows existing Perl operators to be redefined for objects of the current class	5.003
Scalar::Util	Provides additional scalar-processing utilities missing from the core language	5.8
strict	Prohibits unsafe uses of package variables, symbolic references, and barewords	5.8

Module name	Description	Since
Test::Harness	Executes and summarizes Perl test suites	5.8
Test::More	Provides more sophisticated utilities for writing tests	5.8
Time::HiRes	Installs high-resolution versions of Perl's built-in time-keeping functions	5.8
version	Allows multipart versions to be specified as objects	5.10

Recommended CPAN Modules

Module name	Description	Recommended
Attribute::Types	Provides markers that confer type constraints on variables	0.10 or later
Class::Std	Implements encapsulated class hierarchies	Any
Class::Std::Utils	Provides utility functions for producing unique identifiers for any object, for creating anonymous scalars, and for extracting initialization values from a hierarchical initializer list	Any
Config::General	Reads and writes almost any type of configuration file	2.27 or later
Config::Std	Reads and writes simple configuration files, preserving their structure and comments	Any
Config::Tiny	Reads and writes simple "INI" format configuration files with as little code as possible	2.01 or later
Contextual::Return	Simplifies returning different values in different contexts	Any
Data::Alias	Provides a comprehensive set of operations for aliasing variables	0.04 or later
DateTime	Creates powerful date and time objects	0.28 or later
DBI	Provides a generic interface to a large number of databases (see also the many DBD:: modules)	1.48 or later
Devel::Size	Reports the amount of memory a variable uses	0.59 or later

Module name	Description	Recommended
Exception::Class	Simplifies the creation of exception class hierarchies	1.20 or later
File::Slurp	Permits efficient reading and writing of entire files	Any
Filter::Macro	Converts a module into a macro that is expanded inline when the module is loaded	0.02 or later
Getopt::Clade	Builds command-line parsers from an WYSIWYG declaration	Any
Getopt::Euclid	Builds command-line parsers from command-line documentation	Any
HTML::Mason	Builds web sites from modular Perl/HTML specifications	1.28 or later
Inline	Enables Perl subroutines to be written in other programming languages	0.44 or later
IO::InSitu	Allows a file to be modified in place with backup protection	Any
IO::Interactive	Supplies handy subroutines for testing interactivity	Any
IO::Prompt	Simplifies interactive prompting for user input	0.02 or later
Lexical::Alias	Provides a smaller set of operations for aliasing variables	0.04 or later
List::Cycle	Creates objects that can cycle through lists of values	Any
List::MoreUtils	Provides additional list-processing utilities missing from the core language and the List::Util module	0.09 or later
Log::Stdlog	Allows simple event logging via a special filehandle	Any
Module::Build	Builds, tests, and installs Perl modules	0.2609 or later
Module::Starter	Creates the directory structures and starter files needed to develop a Perl module	1.38 or later
Module::Starter::PBP	Creates the directory structures and starter files needed to develop a Perl module conforming to the guidelines presented in this	Any

Module name	Description	Recommended
	book	
only	Loads only specific versions of a module	0.27 or later
Parse::RecDescent	Creates recursive-descent parsers	1.94 or later
Perl6::Builtins	Provides updated versions of several Perl builtins, notably the system command, with extra features that will be standard in Perl 6	Any
Perl6::Export::Attrs	Provides a simple and robust way to export subroutines from modules	Any
Perl6::Form	Implements a replacement for Perl format statements	0.04 or later
Perl6::Slurp	Opens a file and reads its contents in one statement	0.03 or later
POE	Implements a portable multitasking and networking framework for Perl	0.3009 or later
Readonly	Creates read-only scalars, arrays, and hashes	1.03 or later
Regexp::Autoflags	Automatically appends /xms to all regexes	Any
Regexp::Assemble	Combines simple patterns into a single complex pattern	0.10 or later
Regexp::Common	Generates many commonly needed regular expressions	2.120 or later
Regexp::MatchContext	Defines "match variables" that aren't prohibitively expensive	Any
Smart::Comments	Enables special comments for debugging and reporting the progress of non-interactive loops	Any
Sort::Maker	Creates efficient sorting subroutines from simple descriptions	Any
Sub::Installer	Installs subroutines in packages	Any
Text::Autoformat	Automatically wraps and reformats plain text based on its content	1.12 or later
Text::CSV	Provides tools to manipulate comma-separated value strings	Any
Text::CSV::Simple	Simplifies parsing of CSV files	0.20 or later

Module name	Description	Recommended
Text::CSV_XS	Provides faster 8-bit-clean tools to manipulate comma separated value strings	0.23 or later
XML::Parser	Parses XML documents using the Expat library	2.34 or later
YAML	Serializes Perl data structures to a compact and readable string representation	0.38

Utility Subroutines

Subroutine	Description	Available
all( )	Returns true if all its arguments are true	List::MoreUtils
anon_scalar( )	Returns a reference to an anonymous scalar	Class::Std::Utils
any( )	Returns true if any of its arguments are true	List::MoreUtils
apply( )	Applies a transformation to its list of arguments	List::MoreUtils
blessed( )	Returns true if its argument is a reference to a blessed object	Scalar::Util
carp( )	Prints a warning like **warn** does, but reports it from the caller's location	Carp
cmp_these( )	Times a set of alternative code fragments and compares the results in a table	Benchmark
croak( )	Throws an exception like **die** does, but reports it from the caller's location	Carp
first( )	Returns the first of its arguments that satisfies some test	List::Util
first_index( )	Returns the index of the first of its arguments that satisfies some test	List::MoreUtils
form( )	Formats data into fixed-field reports	Perl6::Form
ident( )	Returns a unique identifier for an object	Class::Std::Utils
interactive( )	Returns an output filehandle that ignores output unless the current process is interactive	IO::Interactive
is( )	Tests whether its arguments are equal and reports accordingly	Test::More
is_interactive( )	Returns true if the current process is interactive	IO::Interactive

Subroutine	Description	Available
is_weak( )	Returns true if its argument is a reference that is invisible to the garbage collector	Scalar::Util
looks_like_number( )	Returns true if the argument is something Perl could convert to a number	Scalar::Util
make_sorter( )	Generates efficient sorting routines	Sort::Maker
max( )	Returns the maximum of a list of numbers	List::Util
maxstr( )	Returns the lexicographically last value from a list of strings	List::Util
memoize( )	Causes a subroutine to cache its return values	Memoize
min( )	Returns the minimum of a list of numbers	List::Util
minstr( )	Returns the lexicographically first value from a list of strings	List::Util
none( )	Returns true if none of its arguments is true	List::MoreUtils
notall( )	Returns true if any of its arguments is false	List::MoreUtils
ok( )	Tests whether a condition is true and reports accordingly	Test::Simple
openhandle( )	Returns true if its argument is an open filehandle	Scalar::Util
pairwise( )	Applies a specified binary operation between corresponding elements of two arrays	List::MoreUtils
prompt( )	Prints a prompt, reads some input, verifies it, and returns it	IO::Prompt
qv( )	Creates a version number object	version
readonly( )	Returns true if its argument is not assignable	Scalar::Util
read_file( )	Reads and returns the entire contents of a file	File::Slurp
reduce( )	Applies a specified binary operation between every adjacent element in a list	List::Util
refaddr( )	Returns the address of a reference, as an integer	Scalar::Util
reftype( )	Returns a string representing the underlying type of a reference	Scalar::Util
shuffle( )	Returns its arguments in a (pseudo)random order	List::Util
size( )	Returns the amount of memory used to store the data in its argument	Devel::Size

Subroutine	Description	Available
slurp( )	Reads the entire contents of a file	Perl6::Slurp
sum( )	Returns the numeric sum of its arguments	List::Util
tainted( )	Returns true if its argument is tainted	Scalar::Util
total_size( )	Returns the amount of memory used to store the data and implementation of its argument	Devel::Size
uniq( )	Returns its list of arguments with any duplicates removed	List::MoreUtils
usleep( )	Sleeps a specified number of microseconds	Time::HiRes
weaken( )	Hides a reference from the garbage collector	Scalar::Util
zip( )	Interleaves values from two or more arrays	List::MoreUtils

Utils

CookBook::Utils

```
package CookBook::Utils;

use strict;
use warnings;
use Exporter;
use Term::ReadKey;
use Time::localtime;
use Scalar::Util qw(blessed reftype);

our (@ISA, @EXPORT, $VERSION);

@ISA = ("Exporter");
$VERSION = 1.00;
@EXPORT = qw(
    is_palindrome
    reverse_words
    trim
    ltrim
    rtrim
    print_array
    uniq
    read_key
    read_password
    count_lines
    get_current_date
    get_current_time
    number_with_commas
    round
    sleep_for
    is_object
    ref_type_of
    println
    how_many_defined
    defined_arguments_in
    swap_arrays
    unique_via_anon
    unique_via_grep
    unique_via_slice
    slurp_file
);
```

```
sub is_palindrome {
    my ($input) = @_;
    return ($input eq reverse $input);
}

sub reverse_words {
    my ($input) = @_;
    return join(" ", reverse split(" ", $input));
}

sub trim {
    ltrim(rtrim(@_));
}

sub ltrim {
   my @out = @_;
   for (@out) {
      s/^\s+//;        # trim left
   }
   return @out == 1
        ? $out[0]   # only one to return
        : @out;    # or many
}

sub rtrim {
   my @out = @_;
   for (@out) {
      s/\s+$//;        # trim right
   }
   return @out == 1
        ? $out[0]   # only one to return
        : @out;    # or many
}

sub print_array {
    my $out = (@_ == 0) ? ''
            : (@_ == 1) ? $_[0]
            : (@_ == 2) ? join(" and ", @_)
            :             join(", ", @_[0..($#_ -1)], "and $_[-1]")
            ;
    print $out, "\n";
    return;
}
```

```
sub uniq {
    return unique_via_grep(@_);
}

sub unique_via_anon {
   return keys %{ { map {$_=>1} @_ } };
}

sub unique_via_grep {
   my %seen = ();
   return grep { !$seen{$_}++ } @_;
}

sub unique_via_slice {
   my %uniq;
   @uniq{@_} = ();
   return keys %uniq;
}


sub UNIVERSAL::copy {
   my $self = shift;
   unless (ref $self) {
      require Carp;
      Carp::croak("can't copy class $self");
   }
   require Storable;
   my $copy = Storable::dclone($self);
   return $copy;
}

sub read_key {
    ReadMode 'cbreak';
    my $key = ReadKey(0);
    ReadMode 'normal';
    return $key;
}

sub read_password {
    print "Enter your password: ";
    ReadMode 'noecho';
    my $password = ReadLine(0);
    chomp $password;
    ReadMode 'normal';
```

```
    print "\n";
    return $password;
}

sub count_lines {
    open(FILE, "<", shift) or die "can't open $_[0]: $!";
    my $count = 0;
    $count++ while <FILE>;
    close FILE;
    return $count;
}

sub get_current_date {
    my $tm = localtime;
    return sprintf("%04d-%02d-%02d", $tm->year + 1900, $tm->mon + 1, $tm->mday);
}

sub get_current_time {
    my $tm = localtime;
    return sprintf("%04d-%02d-%02d %02d:%02d:%02d", $tm->year + 1900, $tm->mon
+ 1, $tm->mday, $tm->hour, $tm->min, $tm->sec);
}

sub number_with_commas {
    my $text = reverse $_[0];
   $text =~ s/(\d\d\d)(?=\d)(?!\d*\.)/$1,/g;
   return scalar reverse $text;
}

sub sleep_for {
    my $duration = shift;
    select undef, undef, undef, $duration;
    return;
}

sub is_object {
    return blessed(shift);
}

sub ref_type_of {
    return reftype(shift);
}

sub println {
```

```
    return print @_, "\n";
}

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

sub defined_arguments_in {
    return grep {defined $_} @_;
}

#sub swap_arrays(\@\@) {
#   my ($array1_ref, $array2_ref) = @_;
#
#   my @temp_array = @{$array1_ref};
#   @{$array1_ref} = @{$array2_ref};
#   @{$array2_ref} = @temp_array;
#
#   return;
#}

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

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

    return;
}

sub slurp_file {
    open my $in, '<', shift;
    my $text = do { local $/; <$in> };
    close $in;
    return $text;
}


1;
```

UnitTest.pl

```
use strict;
use warnings;
```

```
use Test::More "no_plan";

use lib ".";
use CookBook::Utils;

ok(is_palindrome("deedeed"), "palindrome");
ok(is_palindrome("degged"), "palindrome");

is(reverse_words(
    'Yoda said, "can you see this?"'),
    'this?" see you "can said, Yoda',
    "reverse_words");

is(trim("  \t abc   \t "), "abc", "trim");
is(ltrim("  \t abc"), "abc", "ltrim");
is(rtrim("abc   \t "), "abc", "rtrim");


is(scalar uniq(1, 1, 2, 4, 5, 5), 4, "uniq");

is(number_with_commas(1234567), "1,234,567", "number_with_commas");
is(number_with_commas(12345679), "12,345,679", "number_with_commas");
```