

## NAME

perlfaq4 - Data Manipulation

## DESCRIPTION

This section of the FAQ answers questions related to manipulating numbers, dates, strings, arrays, hashes, and miscellaneous data issues.

### Data: Numbers

#### Why am I getting long decimals (eg, 19.9499999999999) instead of the numbers I should be getting (eg, 19.95)?

For the long explanation, see David Goldberg's "What Every Computer Scientist Should Know About Floating-Point Arithmetic" (<http://web.cse.msu.edu/~cse320/Documents/FloatingPoint.pdf>).

Internally, your computer represents floating-point numbers in binary. Digital (as in powers of two) computers cannot store all numbers exactly. Some real numbers lose precision in the process. This is a problem with how computers store numbers and affects all computer languages, not just Perl.

*perlnumber* shows the gory details of number representations and conversions.

To limit the number of decimal places in your numbers, you can use the `printf` or `sprintf` function. See *"Floating-point Arithmetic" in perlfaq* for more details.

```
printf "%.2f", 10/3;
```

```
my $number = sprintf "%.2f", 10/3;
```

#### Why is `int()` broken?

Your `int()` is most probably working just fine. It's the numbers that aren't quite what you think.

First, see the answer to "Why am I getting long decimals (eg, 19.9499999999999) instead of the numbers I should be getting (eg, 19.95)?".

For example, this

```
print int(0.6/0.2-2), "\n";
```

will in most computers print 0, not 1, because even such simple numbers as 0.6 and 0.2 cannot be presented exactly by floating-point numbers. What you think in the above as 'three' is really more like 2.9999999999999995559.

#### Why isn't my octal data interpreted correctly?

(contributed by brian d foy)

You're probably trying to convert a string to a number, which Perl only converts as a decimal number. When Perl converts a string to a number, it ignores leading spaces and zeroes, then assumes the rest of the digits are in base 10:

```
my $string = '0644';
```

```
print $string + 0; # prints 644
```

```
print $string + 44; # prints 688, certainly not octal!
```

This problem usually involves one of the Perl built-ins that has the same name a Unix command that uses octal numbers as arguments on the command line. In this example, `chmod` on the command line knows that its first argument is octal because that's what it does:

```
%prompt> chmod 644 file
```

If you want to use the same literal digits (644) in Perl, you have to tell Perl to treat them as octal numbers either by prefixing the digits with a 0 or using `oct`:

```
chmod(      0644, $filename ); # right, has leading zero
chmod( oct(644), $filename ); # also correct
```

The problem comes in when you take your numbers from something that Perl thinks is a string, such as a command line argument in `@ARGV`:

```
chmod( $ARGV[0],      $filename ); # wrong, even if "0644"

chmod( oct($ARGV[0]), $filename ); # correct, treat string as octal
```

You can always check the value you're using by printing it in octal notation to ensure it matches what you think it should be. Print it in octal and decimal format:

```
printf "0%o %d", $number, $number;
```

### Does Perl have a `round()` function? What about `ceil()` and `floor()`? Trig functions?

Remember that `int()` merely truncates toward 0. For rounding to a certain number of digits, `sprintf()` or `printf()` is usually the easiest route.

```
printf("%.3f", 3.1415926535); # prints 3.142
```

The *POSIX* module (part of the standard Perl distribution) implements `ceil()`, `floor()`, and a number of other mathematical and trigonometric functions.

```
use POSIX;
my $ceil = ceil(3.5); # 4
my $floor = floor(3.5); # 3
```

In 5.000 to 5.003 perls, trigonometry was done in the *Math::Complex* module. With 5.004, the *Math::Trig* module (part of the standard Perl distribution) implements the trigonometric functions. Internally it uses the *Math::Complex* module and some functions can break out from the real axis into the complex plane, for example the inverse sine of 2.

Rounding in financial applications can have serious implications, and the rounding method used should be specified precisely. In these cases, it probably pays not to trust whichever system of rounding is being used by Perl, but instead to implement the rounding function you need yourself.

To see why, notice how you'll still have an issue on half-way-point alternation:

```
for (my $i = 0; $i < 1.01; $i += 0.05) { printf "%.1f ", $i }

0.0 0.1 0.1 0.2 0.2 0.2 0.3 0.3 0.4 0.4 0.5 0.5 0.6 0.7 0.7
0.8 0.8 0.9 0.9 1.0 1.0
```

Don't blame Perl. It's the same as in C. IEEE says we have to do this. Perl numbers whose absolute values are integers under  $2^{31}$  (on 32-bit machines) will work pretty much like mathematical integers. Other numbers are not guaranteed.

## How do I convert between numeric representations/bases/radixes?

As always with Perl there is more than one way to do it. Below are a few examples of approaches to making common conversions between number representations. This is intended to be representational rather than exhaustive.

Some of the examples later in *perlfaq4* use the *Bit::Vector* module from CPAN. The reason you might choose *Bit::Vector* over the perl built-in functions is that it works with numbers of ANY size, that it is optimized for speed on some operations, and for at least some programmers the notation might be familiar.

### How do I convert hexadecimal into decimal

Using perl's built in conversion of 0x notation:

```
my $dec = 0xDEADBEEF;
```

Using the `hex` function:

```
my $dec = hex("DEADBEEF");
```

Using `pack`:

```
my $dec = unpack("N", pack("H8", substr("0" x 8 . "DEADBEEF",  
-8)));
```

Using the CPAN module *Bit::Vector*:

```
use Bit::Vector;  
my $vec = Bit::Vector->new_Hex(32, "DEADBEEF");  
my $dec = $vec->to_Dec();
```

### How do I convert from decimal to hexadecimal

Using `sprintf`:

```
my $hex = sprintf("%X", 3735928559); # upper case A-F  
my $hex = sprintf("%x", 3735928559); # lower case a-f
```

Using `unpack`:

```
my $hex = unpack("H*", pack("N", 3735928559));
```

Using *Bit::Vector*:

```
use Bit::Vector;  
my $vec = Bit::Vector->new_Dec(32, -559038737);  
my $hex = $vec->to_Hex();
```

And *Bit::Vector* supports odd bit counts:

```
use Bit::Vector;  
my $vec = Bit::Vector->new_Dec(33, 3735928559);  
$vec->Resize(32); # suppress leading 0 if unwanted  
my $hex = $vec->to_Hex();
```

### How do I convert from octal to decimal

Using Perl's built in conversion of numbers with leading zeros:

```
my $dec = 033653337357; # note the leading 0!
```

Using the `oct` function:

```
my $dec = oct("33653337357");
```

Using *Bit::Vector*:

```
use Bit::Vector;
my $vec = Bit::Vector->new(32);
$vec->Chunk_List_Store(3, split(//, reverse "33653337357"));
my $dec = $vec->to_Dec();
```

How do I convert from decimal to octal

Using `sprintf`:

```
my $oct = sprintf("%o", 3735928559);
```

Using *Bit::Vector*:

```
use Bit::Vector;
my $vec = Bit::Vector->new_Dec(32, -559038737);
my $oct = reverse join(' ', $vec->Chunk_List_Read(3));
```

How do I convert from binary to decimal

Perl 5.6 lets you write binary numbers directly with the `0b` notation:

```
my $number = 0b10110110;
```

Using `oct`:

```
my $input = "10110110";
my $decimal = oct( "0b$input" );
```

Using `pack` and `ord`:

```
my $decimal = ord(pack('B8', '10110110'));
```

Using `pack` and `unpack` for larger strings:

```
my $int = unpack("N", pack("B32",
    substr("0" x 32 . "11110101011011011111011101111", -32)));
my $dec = sprintf("%d", $int);
```

# `substr()` is used to left-pad a 32-character string with zeros.

Using *Bit::Vector*:

```
my $vec = Bit::Vector->new_Bin(32,
    "11011110101011011011011111011101111");
my $dec = $vec->to_Dec();
```

How do I convert from decimal to binary

Using `sprintf` (perl 5.6+):

```
my $bin = sprintf("%b", 3735928559);
```

Using `unpack`:

```
my $bin = unpack("B*", pack("N", 3735928559));
```

Using *Bit::Vector*:

```
use Bit::Vector;
my $vec = Bit::Vector->new_Dec(32, -559038737);
my $bin = $vec->to_Bin();
```

The remaining transformations (e.g. hex -> oct, bin -> hex, etc.) are left as an exercise to the inclined reader.

### Why doesn't & work the way I want it to?

The behavior of binary arithmetic operators depends on whether they're used on numbers or strings. The operators treat a string as a series of bits and work with that (the string "3" is the bit pattern 00110011). The operators work with the binary form of a number (the number 3 is treated as the bit pattern 00000011).

So, saying `11 & 3` performs the "and" operation on numbers (yielding 3). Saying `"11" & "3"` performs the "and" operation on strings (yielding "1").

Most problems with `&` and `|` arise because the programmer thinks they have a number but really it's a string or vice versa. To avoid this, stringify the arguments explicitly (using `"` or `qq()`) or convert them to numbers explicitly (using `0+$arg`). The rest arise because the programmer says:

```
if ("\020\020" & "\101\101") {
    # ...
}
```

but a string consisting of two null bytes (the result of `"\020\020" & "\101\101"`) is not a false value in Perl. You need:

```
if ( ("\020\020" & "\101\101") !~ /[^\000]/) {
    # ...
}
```

### How do I multiply matrices?

Use the *Math::Matrix* or *Math::MatrixReal* modules (available from CPAN) or the *PDL* extension (also available from CPAN).

### How do I perform an operation on a series of integers?

To call a function on each element in an array, and collect the results, use:

```
my @results = map { my_func($_) } @array;
```

For example:

```
my @triple = map { 3 * $_ } @single;
```

To call a function on each element of an array, but ignore the results:

```
foreach my $iterator (@array) {
    some_func($iterator);
}
```

To call a function on each integer in a (small) range, you **can** use:

```
my @results = map { some_func($_) } (5 .. 25);
```

but you should be aware that in this form, the `..` operator creates a list of all integers in the range, which can take a lot of memory for large ranges. However, the problem does not occur when using `..` within a `for` loop, because in that case the range operator is optimized to *iterate* over the range, without creating the entire list. So

```
my @results = ();
for my $i (5 .. 500_005) {
```

```
        push(@results, some_func($i));
    }
```

or even

```
push(@results, some_func($_)) for 5 .. 500_005;
```

will not create an intermediate list of 500,000 integers.

## How can I output Roman numerals?

Get the <http://www.cpan.org/modules/by-module/Roman> module.

## Why aren't my random numbers random?

If you're using a version of Perl before 5.004, you must call `srand` once at the start of your program to seed the random number generator.

```
BEGIN { srand() if $] < 5.004 }
```

5.004 and later automatically call `srand` at the beginning. Don't call `srand` more than once--you make your numbers less random, rather than more.

Computers are good at being predictable and bad at being random (despite appearances caused by bugs in your programs :-). The *random* article in the "Far More Than You Ever Wanted To Know" collection in <http://www.cpan.org/misc/olddoc/FMTEYEWTK.tgz>, courtesy of Tom Phoenix, talks more about this. John von Neumann said, "Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin."

Perl relies on the underlying system for the implementation of `rand` and `srand`; on some systems, the generated numbers are not random enough (especially on Windows : see [http://www.perlmonks.org/?node\\_id=803632](http://www.perlmonks.org/?node_id=803632)). Several CPAN modules in the `Math` namespace implement better pseudorandom generators; see for example `Math::Random::MT` ("Mersenne Twister", fast), or `Math::TrulyRandom` (uses the imperfections in the system's timer to generate random numbers, which is rather slow). More algorithms for random numbers are described in "Numerical Recipes in C" at <http://www.nr.com/>

## How do I get a random number between X and Y?

To get a random number between two values, you can use the `rand()` built-in to get a random number between 0 and 1. From there, you shift that into the range that you want.

`rand($x)` returns a number such that  $0 \leq \text{rand}(\$x) < \$x$ . Thus what you want to have perl figure out is a random number in the range from 0 to the difference between your X and Y.

That is, to get a number between 10 and 15, inclusive, you want a random number between 0 and 5 that you can then add to 10.

```
my $number = 10 + int rand( 15-10+1 ); # ( 10,11,12,13,14, or 15 )
```

Hence you derive the following simple function to abstract that. It selects a random integer between the two given integers (inclusive), For example: `random_int_between(50,120)`.

```
sub random_int_between {
    my($min, $max) = @_;
    # Assumes that the two arguments are integers themselves!
    return $min if $min == $max;
    ($min, $max) = ($max, $min) if $min > $max;
    return $min + int rand(1 + $max - $min);
}
```

## Data: Dates

### How do I find the day or week of the year?

The day of the year is in the list returned by the `localtime` function. Without an argument `localtime` uses the current time.

```
my $day_of_year = (localtime)[7];
```

The *POSIX* module can also format a date as the day of the year or week of the year.

```
use POSIX qw/strftime/;
my $day_of_year = strftime "%j", localtime;
my $week_of_year = strftime "%W", localtime;
```

To get the day of year for any date, use *POSIX*'s `mktime` to get a time in epoch seconds for the argument to `localtime`.

```
use POSIX qw/mktime strftime/;
my $week_of_year = strftime "%W",
    localtime( mktime( 0, 0, 0, 18, 11, 87 ) );
```

You can also use *Time::Piece*, which comes with Perl and provides a `localtime` that returns an object:

```
use Time::Piece;
my $day_of_year = localtime->yday;
my $week_of_year = localtime->week;
```

The *Date::Calc* module provides two functions to calculate these, too:

```
use Date::Calc;
my $day_of_year = Day_of_Year( 1987, 12, 18 );
my $week_of_year = Week_of_Year( 1987, 12, 18 );
```

### How do I find the current century or millennium?

Use the following simple functions:

```
sub get_century {
    return int((((localtime(shift || time))[5] + 1999))/100);
}

sub get_millennium {
    return 1+int((((localtime(shift || time))[5] + 1899))/1000);
}
```

On some systems, the *POSIX* module's `strftime()` function has been extended in a non-standard way to use a `%C` format, which they sometimes claim is the "century". It isn't, because on most such systems, this is only the first two digits of the four-digit year, and thus cannot be used to determine reliably the current century or millennium.

### How can I compare two dates and find the difference?

(contributed by brian d foy)

You could just store all your dates as a number and then subtract. Life isn't always that simple though.

The *Time::Piece* module, which comes with Perl, replaces *localtime* with a version that returns an object. It also overloads the comparison operators so you can compare them directly:

```
use Time::Piece;
my $date1 = localtime( $some_time );
my $date2 = localtime( $some_other_time );

if( $date1 < $date2 ) {
    print "The date was in the past\n";
}
```

You can also get differences with a subtraction, which returns a *Time::Seconds* object:

```
my $diff = $date1 - $date2;
print "The difference is ", $date_diff->days, " days\n";
```

If you want to work with formatted dates, the *Date::Manip*, *Date::Calc*, or *DateTime* modules can help you.

### How can I take a string and turn it into epoch seconds?

If it's a regular enough string that it always has the same format, you can split it up and pass the parts to *timelocal* in the standard *Time::Local* module. Otherwise, you should look into the *Date::Calc*, *Date::Parse*, and *Date::Manip* modules from CPAN.

### How can I find the Julian Day?

(contributed by brian d foy and Dave Cross)

You can use the *Time::Piece* module, part of the Standard Library, which can convert a date/time to a Julian Day:

```
$ perl -MTime::Piece -le 'print localtime->julian_day'
2455607.7959375
```

Or the modified Julian Day:

```
$ perl -MTime::Piece -le 'print localtime->mjd'
55607.2961226851
```

Or even the day of the year (which is what some people think of as a Julian day):

```
$ perl -MTime::Piece -le 'print localtime->yday'
45
```

You can also do the same things with the *DateTime* module:

```
$ perl -MDateTime -le'print DateTime->today->jd'
2453401.5
$ perl -MDateTime -le'print DateTime->today->mjd'
53401
$ perl -MDateTime -le'print DateTime->today->doy'
31
```

You can use the *Time::JulianDay* module available on CPAN. Ensure that you really want to find a Julian day, though, as many people have different ideas about Julian days (see [http://www.hermetic.ch/cal\\_stud/jdn.htm](http://www.hermetic.ch/cal_stud/jdn.htm) for instance):



```
$ perl -MTime::JulianDay -le 'print local_julian_day( time )'
55608
```

## How do I find yesterday's date?

(contributed by brian d foy)

To do it correctly, you can use one of the `Date` modules since they work with calendars instead of times. The `DateTime` module makes it simple, and give you the same time of day, only the day before, despite daylight saving time changes:

```
use DateTime;

my $yesterday = DateTime->now->subtract( days => 1 );

print "Yesterday was $yesterday\n";
```

You can also use the `Date::Calc` module using its `Today_and_Now` function.

```
use Date::Calc qw( Today_and_Now Add_Delta_DHMS );

my @date_time = Add_Delta_DHMS( Today_and_Now(), -1, 0, 0, 0 );

print "@date_time\n";
```

Most people try to use the time rather than the calendar to figure out dates, but that assumes that days are twenty-four hours each. For most people, there are two days a year when they aren't: the switch to and from summer time throws this off. For example, the rest of the suggestions will be wrong sometimes:

Starting with Perl 5.10, `Time::Piece` and `Time::Seconds` are part of the standard distribution, so you might think that you could do something like this:

```
use Time::Piece;
use Time::Seconds;

my $yesterday = localtime() - ONE_DAY; # WRONG
print "Yesterday was $yesterday\n";
```

The `Time::Piece` module exports a new `localtime` that returns an object, and `Time::Seconds` exports the `ONE_DAY` constant that is a set number of seconds. This means that it always gives the time 24 hours ago, which is not always yesterday. This can cause problems around the end of daylight saving time when there's one day that is 25 hours long.

You have the same problem with `Time::Local`, which will give the wrong answer for those same special cases:

```
# contributed by Gunnar Hjalmarsson
use Time::Local;
my $today = timelocal 0, 0, 12, ( localtime )[3..5];
my ($d, $m, $y) = ( localtime $today-86400 )[3..5]; # WRONG
printf "Yesterday: %d-%02d-%02d\n", $y+1900, $m+1, $d;
```

## Does Perl have a Year 2000 or 2038 problem? Is Perl Y2K compliant?

(contributed by brian d foy)

Perl itself never had a Y2K problem, although that never stopped people from creating Y2K problems on their own. See the documentation for `localtime` for its proper use.

Starting with Perl 5.12, `localtime` and `gmtime` can handle dates past 03:14:08 January 19, 2038, when a 32-bit based time would overflow. You still might get a warning on a 32-bit `perl`:

```
% perl5.12 -E 'say scalar localtime( 0x9FFF_FFFFFFFF )'
Integer overflow in hexadecimal number at -e line 1.
Wed Nov  1 19:42:39 5576711
```

On a 64-bit `perl`, you can get even larger dates for those really long running projects:

```
% perl5.12 -E 'say scalar gmtime( 0x9FFF_FFFFFFFF )'
Thu Nov  2 00:42:39 5576711
```

You're still out of luck if you need to keep track of decaying protons though.

## Data: Strings

### How do I validate input?

(contributed by brian d foy)

There are many ways to ensure that values are what you expect or want to accept. Besides the specific examples that we cover in the *perlfaq*, you can also look at the modules with "Assert" and "Validate" in their names, along with other modules such as *Regexp::Common*.

Some modules have validation for particular types of input, such as *Business::ISBN*, *Business::CreditCard*, *Email::Valid*, and *Data::Validate::IP*.

### How do I unescape a string?

It depends just what you mean by "escape". URL escapes are dealt with in *perlfaq9*. Shell escapes with the backslash (`\`) character are removed with

```
s/\\(\\.)/$1/g;
```

This won't expand "`\n`" or "`\t`" or any other special escapes.

### How do I remove consecutive pairs of characters?

(contributed by brian d foy)

You can use the substitution operator to find pairs of characters (or runs of characters) and replace them with a single instance. In this substitution, we find a character in `(. )`. The memory parentheses store the matched character in the back-reference `\g1` and we use that to require that the same thing immediately follow it. We replace that part of the string with the character in `$1`.

```
s/(. )\g1/$1/g;
```

We can also use the transliteration operator, `tr///`. In this example, the search list side of our `tr///` contains nothing, but the `c` option complements that so it contains everything. The replacement list also contains nothing, so the transliteration is almost a no-op since it won't do any replacements (or more exactly, replace the character with itself). However, the `s` option squashes duplicated and consecutive characters in the string so a character does not show up next to itself

```
my $str = 'Haarlem';    # in the Netherlands
$str =~ tr///cs;        # Now Harlem, like in New York
```

## How do I expand function calls in a string?

(contributed by brian d foy)

This is documented in *perlref*, and although it's not the easiest thing to read, it does work. In each of these examples, we call the function inside the braces used to dereference a reference. If we have more than one return value, we can construct and dereference an anonymous array. In this case, we call the function in list context.

```
print "The time values are @{ [localtime] }.\n";
```

If we want to call the function in scalar context, we have to do a bit more work. We can really have any code we like inside the braces, so we simply have to end with the scalar reference, although how you do that is up to you, and you can use code inside the braces. Note that the use of parens creates a list context, so we need *scalar* to force the scalar context on the function:

```
print "The time is ${\(scalar localtime)}.\n"
```

```
print "The time is ${ my $x = localtime; \ $x }.\n";
```

If your function already returns a reference, you don't need to create the reference yourself.

```
sub timestamp { my $t = localtime; \ $t }
```

```
print "The time is ${ timestamp() }.\n";
```

The *Interpolation* module can also do a lot of magic for you. You can specify a variable name, in this case *E*, to set up a tied hash that does the interpolation for you. It has several other methods to do this as well.

```
use Interpolation E => 'eval';
print "The time values are $E{localtime()}.\n";
```

In most cases, it is probably easier to simply use string concatenation, which also forces scalar context.

```
print "The time is " . localtime() . ".\n";
```

## How do I find matching/nesting anything?

To find something between two single characters, a pattern like `/x([^\x]*)x/` will get the intervening bits in *\$1*. For multiple ones, then something more like `/alpha(.*)omega/` would be needed. For nested patterns and/or balanced expressions, see the so-called (*?PARNO*) construct (available since perl 5.10). The CPAN module *Regexp::Common* can help to build such regular expressions (see in particular *Regexp::Common::balanced* and *Regexp::Common::delimited*).

More complex cases will require to write a parser, probably using a parsing module from CPAN, like *Regexp::Grammars*, *Parse::RecDescent*, *Parse::Yapp*, *Text::Balanced*, or *Marpa::XS*.

## How do I reverse a string?

Use `reverse()` in scalar context, as documented in *"reverse" in perlfunc*.

```
my $reversed = reverse $string;
```

## How do I expand tabs in a string?

You can do it yourself:

```
1 while $string =~ s/\t+/' ' x (length($&) * 8 - length($`) % 8)/e;
```

Or you can just use the *Text::Tabs* module (part of the standard Perl distribution).

```
use Text::Tabs;
my @expanded_lines = expand(@lines_with_tabs);
```

## How do I reformat a paragraph?

Use *Text::Wrap* (part of the standard Perl distribution):

```
use Text::Wrap;
print wrap("\t", ' ', @paragraphs);
```

The paragraphs you give to *Text::Wrap* should not contain embedded newlines. *Text::Wrap* doesn't justify the lines (flush-right).

Or use the CPAN module *Text::Autoformat*. Formatting files can be easily done by making a shell alias, like so:

```
alias fmt="perl -i -MText::Autoformat -n0777 \
-e 'print autoformat $_, {all=>1}' $*"
```

See the documentation for *Text::Autoformat* to appreciate its many capabilities.

## How can I access or change N characters of a string?

You can access the first characters of a string with *substr()*. To get the first character, for example, start at position 0 and grab the string of length 1.

```
my $string = "Just another Perl Hacker";
my $first_char = substr( $string, 0, 1 ); # 'J'
```

To change part of a string, you can use the optional fourth argument which is the replacement string.

```
substr( $string, 13, 4, "Perl 5.8.0" );
```

You can also use *substr()* as an lvalue.

```
substr( $string, 13, 4 ) = "Perl 5.8.0";
```

## How do I change the Nth occurrence of something?

You have to keep track of N yourself. For example, let's say you want to change the fifth occurrence of "whoever" or "whomever" into "whosoever" or "whomsoever", case insensitively. These all assume that *\$\_* contains the string to be altered.

```
$count = 0;
s{((whom?)ever)}{
++$count == 5      # is it the 5th?
? "{$2}soever"    # yes, swap
: $1              # renege and leave it there
}ige;
```

In the more general case, you can use the */g* modifier in a *while* loop, keeping count of matches.

```
$WANT = 3;
$count = 0;
$_ = "One fish two fish red fish blue fish";
```

```
while ((/\w+)\s+fish\b/gi) {  
    if (++$count == $WANT) {  
        print "The third fish is a $1 one.\n";  
    }  
}
```

That prints out: "The third fish is a red one." You can also use a repetition count and repeated pattern like this:

```
/(?:\w+\s+fish\s+){2}(\w+)\s+fish/i;
```

## How can I count the number of occurrences of a substring within a string?

There are a number of ways, with varying efficiency. If you want a count of a certain single character (X) within a string, you can use the `tr///` function like so:

```
my $string = "ThisXlineXhasXsomeXx'sXinXit";  
my $count = ($string =~ tr/X//);  
print "There are $count X characters in the string";
```

This is fine if you are just looking for a single character. However, if you are trying to count multiple character substrings within a larger string, `tr///` won't work. What you can do is wrap a `while()` loop around a global pattern match. For example, let's count negative integers:

```
my $string = "-9 55 48 -2 23 -76 4 14 -44";  
my $count = 0;  
while ($string =~ /\-\\d+/g) { $count++ }  
print "There are $count negative numbers in the string";
```

Another version uses a global match in list context, then assigns the result to a scalar, producing a count of the number of matches.

```
my $count = () = $string =~ /\-\\d+/g;
```

## How do I capitalize all the words on one line?

(contributed by brian d foy)

Damian Conway's *Text::Autoformat* handles all of the thinking for you.

```
use Text::Autoformat;  
my $x = "Dr. Strangelove or: How I Learned to Stop ".  
    "Worrying and Love the Bomb";  
  
print $x, "\n";  
for my $style (qw( sentence title highlight )) {  
    print autoformat($x, { case => $style }), "\n";  
}
```

How do you want to capitalize those words?

FRED AND BARNEY'S LODGE	# all uppercase
Fred And Barney's Lodge	# title case
Fred and Barney's Lodge	# highlight case

It's not as easy a problem as it looks. How many words do you think are in there? Wait for it... wait for it.... If you answered 5 you're right. Perl words are groups of `\w+`, but that's not what you want to

capitalize. How is Perl supposed to know not to capitalize that `s` after the apostrophe? You could try a regular expression:

```
$string =~ s/ (
    (^\\w)      #at the beginning of the line
    |          # or
    (\\s\\w)    #preceded by whitespace
)
/\\U$1/xg;

$string =~ s/([\\w']+)/\\u\\L$1/g;
```

Now, what if you don't want to capitalize that "and"? Just use `Text::Autoformat` and get on with the next problem. :)

### How can I split a [character]-delimited string except when inside [character]?

Several modules can handle this sort of parsing--`Text::Balanced`, `Text::CSV`, `Text::CSV_XS`, and `Text::ParseWords`, among others.

Take the example case of trying to split a string that is comma-separated into its different fields. You can't use `split(/,/)` because you shouldn't split if the comma is inside quotes. For example, take a data line like this:

```
SAR001,"","Cimetrix, Inc","Bob Smith","CAM",N,8,1,0,7,"Error, Core
Dumped"
```

Due to the restriction of the quotes, this is a fairly complex problem. Thankfully, we have Jeffrey Friedl, author of *Mastering Regular Expressions*, to handle these for us. He suggests (assuming your string is contained in `$text`):

```
my @new = ();
push(@new, $+) while $text =~ m{
    "([^\\"\\]*(?:\\\"\\\"|\\\\\\\\)*)"?,? # groups the phrase inside the
quotes
    | ([^,]+),?
    | ,
}gx;
push(@new, undef) if substr($text,-1,1) eq ',';
```

If you want to represent quotation marks inside a quotation-mark-delimited field, escape them with backslashes (eg, "like \"this\"").

Alternatively, the `Text::ParseWords` module (part of the standard Perl distribution) lets you say:

```
use Text::ParseWords;
@new = quotewords(",", 0, $text);
```

For parsing or generating CSV, though, using `Text::CSV` rather than implementing it yourself is highly recommended; you'll save yourself odd bugs popping up later by just using code which has already been tried and tested in production for years.

### How do I strip blank space from the beginning/end of a string?

(contributed by brian d foy)

A substitution can do this for you. For a single line, you want to replace all the leading or trailing whitespace with nothing. You can do that with a pair of substitutions:

```
s/^s+//;
s/s+$//;
```

You can also write that as a single substitution, although it turns out the combined statement is slower than the separate ones. That might not matter to you, though:

```
s/^s+|\s+$//g;
```

In this regular expression, the alternation matches either at the beginning or the end of the string since the anchors have a lower precedence than the alternation. With the `/g` flag, the substitution makes all possible matches, so it gets both. Remember, the trailing newline matches the `\s+`, and the `$` anchor can match to the absolute end of the string, so the newline disappears too. Just add the newline to the output, which has the added benefit of preserving "blank" (consisting entirely of whitespace) lines which the `^s+` would remove all by itself:

```
while( <> ) {
    s/^s+|\s+$//g;
    print "$_\n";
}
```

For a multi-line string, you can apply the regular expression to each logical line in the string by adding the `/m` flag (for "multi-line"). With the `/m` flag, the `$` matches *before* an embedded newline, so it doesn't remove it. This pattern still removes the newline at the end of the string:

```
$string =~ s/^s+|\s+$//gm;
```

Remember that lines consisting entirely of whitespace will disappear, since the first part of the alternation can match the entire string and replace it with nothing. If you need to keep embedded blank lines, you have to do a little more work. Instead of matching any whitespace (since that includes a newline), just match the other whitespace:

```
$string =~ s/^[ \t\f ]+|[ \t\f ]+$//mg;
```

## How do I pad a string with blanks or pad a number with zeroes?

In the following examples, `$pad_len` is the length to which you wish to pad the string, `$text` or `$num` contains the string to be padded, and `$pad_char` contains the padding character. You can use a single character string constant instead of the `$pad_char` variable if you know what it is in advance. And in the same way you can use an integer in place of `$pad_len` if you know the pad length in advance.

The simplest method uses the `sprintf` function. It can pad on the left or right with blanks and on the left with zeroes and it will not truncate the result. The `pack` function can only pad strings on the right with blanks and it will truncate the result to a maximum length of `$pad_len`.

```
# Left padding a string with blanks (no truncation):
my $padded = sprintf("%${pad_len}s", $text);
my $padded = sprintf("%*s", $pad_len, $text); # same thing
```

```
# Right padding a string with blanks (no truncation):
my $padded = sprintf("%-${pad_len}s", $text);
my $padded = sprintf("%-*s", $pad_len, $text); # same thing
```

```
# Left padding a number with 0 (no truncation):
my $padded = sprintf("%0${pad_len}d", $num);
my $padded = sprintf("%0*d", $pad_len, $num); # same thing
```

```
# Right padding a string with blanks using pack (will truncate):
my $padded = pack("A$pad_len",$text);
```

If you need to pad with a character other than blank or zero you can use one of the following methods. They all generate a pad string with the `x` operator and combine that with `$text`. These methods do not truncate `$text`.

Left and right padding with any character, creating a new string:

```
my $padded = $pad_char x ( $pad_len - length( $text ) ) . $text;
my $padded = $text . $pad_char x ( $pad_len - length( $text ) );
```

Left and right padding with any character, modifying `$text` directly:

```
substr( $text, 0, 0 ) = $pad_char x ( $pad_len - length( $text ) );
$text .= $pad_char x ( $pad_len - length( $text ) );
```

## How do I extract selected columns from a string?

(contributed by brian d foy)

If you know the columns that contain the data, you can use `substr` to extract a single column.

```
my $column = substr( $line, $start_column, $length );
```

You can use `split` if the columns are separated by whitespace or some other delimiter, as long as whitespace or the delimiter cannot appear as part of the data.

```
my $line      = ' fred barney  betty  ';
my @columns = split /\s+/, $line;
             # ( '', 'fred', 'barney', 'betty' );

my $line      = 'fred||barney||betty';
my @columns = split /\|/, $line;
             # ( 'fred', '', 'barney', '', 'betty' );
```

If you want to work with comma-separated values, don't do this since that format is a bit more complicated. Use one of the modules that handle that format, such as `Text::CSV`, `Text::CSV_XS`, or `Text::CSV_PP`.

If you want to break apart an entire line of fixed columns, you can use `unpack` with the `A` (ASCII) format. By using a number after the format specifier, you can denote the column width. See the `pack` and `unpack` entries in *perlfunc* for more details.

```
my @fields = unpack( $line, "A8 A8 A8 A16 A4" );
```

Note that spaces in the format argument to `unpack` do not denote literal spaces. If you have space separated data, you may want `split` instead.

## How do I find the soundex value of a string?

(contributed by brian d foy)

You can use the `Text::Soundex` module. If you want to do fuzzy or close matching, you might also try the `String::Approx`, and `Text::Metaphone`, and `Text::DoubleMetaphone` modules.



## How can I expand variables in text strings?

(contributed by brian d foy)

If you can avoid it, don't, or if you can use a templating system, such as *Text::Template* or *Template Toolkit*, do that instead. You might even be able to get the job done with `sprintf` or `printf`:

```
my $string = sprintf 'Say hello to %s and %s', $foo, $bar;
```

However, for the one-off simple case where I don't want to pull out a full templating system, I'll use a string that has two Perl scalar variables in it. In this example, I want to expand `$foo` and `$bar` to their variable's values:

```
my $foo = 'Fred';
my $bar = 'Barney';
$string = 'Say hello to $foo and $bar';
```

One way I can do this involves the substitution operator and a double `/e` flag. The first `/e` evaluates `$1` on the replacement side and turns it into `$foo`. The second `/e` starts with `$foo` and replaces it with its value. `$foo`, then, turns into 'Fred', and that's finally what's left in the string:

```
$string =~ s/(\$(\w+))/\1/ee; # 'Say hello to Fred and Barney'
```

The `/e` will also silently ignore violations of strict, replacing undefined variable names with the empty string. Since I'm using the `/e` flag (twice even!), I have all of the same security problems I have with `eval` in its string form. If there's something odd in `$foo`, perhaps something like `@{[ system "rm -rf /" ]}`, then I could get myself in trouble.

To get around the security problem, I could also pull the values from a hash instead of evaluating variable names. Using a single `/e`, I can check the hash to ensure the value exists, and if it doesn't, I can replace the missing value with a marker, in this case `???` to signal that I missed something:

```
my $string = 'This has $foo and $bar';

my %Replacements = (
    foo => 'Fred',
);

# $string =~ s/\$(\w+)/$Replacements{$1}/g;
$string =~ s/\$(\w+)/
    exists $Replacements{$1} ? $Replacements{$1} : '???'
    /eg;

print $string;
```

## What's wrong with always quoting "\$vars"?

The problem is that those double-quotes force stringification--coercing numbers and references into strings--even when you don't want them to be strings. Think of it this way: double-quote expansion is used to produce new strings. If you already have a string, why do you need more?

If you get used to writing odd things like these:

```
print "$var";          # BAD
my $new = "$old";      # BAD
somefunc("$var");      # BAD
```

You'll be in trouble. Those should (in 99.8% of the cases) be the simpler and more direct:

```
print $var;
my $new = $old;
somefunc($var);
```

Otherwise, besides slowing you down, you're going to break code when the thing in the scalar is actually neither a string nor a number, but a reference:

```
func(\@array);
sub func {
    my $aref = shift;
    my $oref = "$aref"; # WRONG
}
```

You can also get into subtle problems on those few operations in Perl that actually do care about the difference between a string and a number, such as the magical ++ autoincrement operator or the syscall() function.

Stringification also destroys arrays.

```
my @lines = `command`;
print "@lines";      # WRONG - extra blanks
print @lines;        # right
```

### Why don't my <<HERE documents work?

Here documents are found in *perlop*. Check for these three things:

There must be no space after the << part.

There (probably) should be a semicolon at the end of the opening token

You can't (easily) have any space in front of the tag.

There needs to be at least a line separator after the end token.

If you want to indent the text in the here document, you can do this:

```
# all in one
(my $VAR = <<HERE_TARGET) =~ s/^\s+//gm;
    your text
    goes here
HERE_TARGET
```

But the HERE\_TARGET must still be flush against the margin. If you want that indented also, you'll have to quote in the indentation.

```
(my $quote = <<'    FINIS') =~ s/^\s+//gm;
    ...we will have peace, when you and all your works have
    perished--and the works of your dark master to whom you
    would deliver us. You are a liar, Saruman, and a corrupter
    of men's hearts. --Theoden in /usr/src/perl/taint.c
    FINIS
$quote =~ s/\s+---/\n---/;
```

A nice general-purpose fixer-upper function for indented here documents follows. It expects to be called with a here document as its argument. It looks to see whether each line begins with a common substring, and if so, strips that substring off. Otherwise, it takes the amount of leading whitespace found on the first line and removes that much off each subsequent line.

```
sub fix {
    local $_ = shift;
    my ($white, $leader); # common whitespace and common leading
string
    if (/^\s*(?:([^\w\s]+)(\s*)).*\n(?:\s*\g1\g2?.*\n)+$/ ) {
        ($white, $leader) = ($2, quotemeta($1));
    } else {
        ($white, $leader) = (/^\s+/ , '');
    }
    s/^\s*?$leader(?:$white)?//gm;
    return $_;
}
```

This works with leading special strings, dynamically determined:

```
my $remember_the_main = fix<<'    MAIN_INTERPRETER_LOOP';
@@@ int
@@@ runops() {
@@@     SAVEI32(runlevel);
@@@     runlevel++;
@@@     while ( op = (*op->op_ppaddr)() );
@@@     TAIINT_NOT;
@@@     return 0;
@@@ }
MAIN_INTERPRETER_LOOP
```

Or with a fixed amount of leading whitespace, with remaining indentation correctly preserved:

```
my $poem = fix<<EVER_ON_AND_ON;
    Now far ahead the Road has gone,
    And I must follow, if I can,
    Pursuing it with eager feet,
    Until it joins some larger way
    Where many paths and errands meet.
    And whither then? I cannot say.
    --Bilbo in /usr/src/perl/pp_ctl.c
EVER_ON_AND_ON
```

## Data: Arrays

### What is the difference between a list and an array?

(contributed by brian d foy)

A list is a fixed collection of scalars. An array is a variable that holds a variable collection of scalars. An array can supply its collection for list operations, so list operations also work on arrays:

```
# slices
( 'dog', 'cat', 'bird' )[2,3];
@animals[2,3];

# iteration
foreach ( qw( dog cat bird ) ) { ... }
foreach ( @animals ) { ... }

my @three = grep { length == 3 } qw( dog cat bird );
my @three = grep { length == 3 } @animals;
```

```
# supply an argument list
wash_animals( qw( dog cat bird ) );
wash_animals( @animals );
```

Array operations, which change the scalars, rearrange them, or add or subtract some scalars, only work on arrays. These can't work on a list, which is fixed. Array operations include `shift`, `unshift`, `push`, `pop`, and `splice`.

An array can also change its length:

```
$#animals = 1; # truncate to two elements
$#animals = 10000; # pre-extend to 10,001 elements
```

You can change an array element, but you can't change a list element:

```
$animals[0] = 'Rottweiler';
qw( dog cat bird )[0] = 'Rottweiler'; # syntax error!

foreach ( @animals ) {
    s/^d/fr/; # works fine
}

foreach ( qw( dog cat bird ) ) {
    s/^d/fr/; # Error! Modification of read only value!
}
```

However, if the list element is itself a variable, it appears that you can change a list element. However, the list element is the variable, not the data. You're not changing the list element, but something the list element refers to. The list element itself doesn't change: it's still the same variable.

You also have to be careful about context. You can assign an array to a scalar to get the number of elements in the array. This only works for arrays, though:

```
my $count = @animals; # only works with arrays
```

If you try to do the same thing with what you think is a list, you get a quite different result. Although it looks like you have a list on the righthand side, Perl actually sees a bunch of scalars separated by a comma:

```
my $scalar = ( 'dog', 'cat', 'bird' ); # $scalar gets bird
```

Since you're assigning to a scalar, the righthand side is in scalar context. The comma operator (yes, it's an operator!) in scalar context evaluates its lefthand side, throws away the result, and evaluates its righthand side and returns the result. In effect, that list-lookalike assigns to `$scalar` its rightmost value. Many people mess this up because they choose a list-lookalike whose last element is also the count they expect:

```
my $scalar = ( 1, 2, 3 ); # $scalar gets 3, accidentally
```

## What is the difference between `$array[1]` and `@array[1]`?

(contributed by brian d foy)

The difference is the sigil, that special character in front of the array name. The `$` sigil means "exactly one item", while the `@` sigil means "zero or more items". The `$` gets you a single scalar, while the `@` gets you a list.

The confusion arises because people incorrectly assume that the sigil denotes the variable type.

The `$array[1]` is a single-element access to the array. It's going to return the item in index 1 (or `undef` if there is no item there). If you intend to get exactly one element from the array, this is the form you should use.

The `@array[1]` is an array slice, although it has only one index. You can pull out multiple elements simultaneously by specifying additional indices as a list, like `@array[1,4,3,0]`.

Using a slice on the lefthand side of the assignment supplies list context to the righthand side. This can lead to unexpected results. For instance, if you want to read a single line from a filehandle, assigning to a scalar value is fine:

```
$array[1] = <STDIN>;
```

However, in list context, the line input operator returns all of the lines as a list. The first line goes into `@array[1]` and the rest of the lines mysteriously disappear:

```
@array[1] = <STDIN>; # most likely not what you want
```

Either the `use warnings` pragma or the `-w` flag will warn you when you use an array slice with a single index.

## How can I remove duplicate elements from a list or array?

(contributed by brian d foy)

Use a hash. When you think the words "unique" or "duplicated", think "hash keys".

If you don't care about the order of the elements, you could just create the hash then extract the keys. It's not important how you create that hash: just that you use `keys` to get the unique elements.

```
my %hash = map { $_, 1 } @array;
# or a hash slice: @hash{ @array } = ();
# or a foreach: $hash{$_} = 1 foreach ( @array );

my @unique = keys %hash;
```

If you want to use a module, try the `uniq` function from *List::MoreUtils*. In list context it returns the unique elements, preserving their order in the list. In scalar context, it returns the number of unique elements.

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

my @unique = uniq( 1, 2, 3, 4, 4, 5, 6, 5, 7 ); # 1,2,3,4,5,6,7
my $unique = uniq( 1, 2, 3, 4, 4, 5, 6, 5, 7 ); # 7
```

You can also go through each element and skip the ones you've seen before. Use a hash to keep track. The first time the loop sees an element, that element has no key in `%Seen`. The `next` statement creates the key and immediately uses its value, which is `undef`, so the loop continues to the `push` and increments the value for that key. The next time the loop sees that same element, its key exists in the hash *and* the value for that key is true (since it's not 0 or `undef`), so the next skips that iteration and the loop goes to the next element.

```
my @unique = ();
my %seen = ();

foreach my $elem ( @array ) {
```

```
    next if $seen{ $elem }++;  
    push @unique, $elem;  
}
```

You can write this more briefly using a `grep`, which does the same thing.

```
my %seen = ();  
my @unique = grep { ! $seen{ $_ }++ } @array;
```

## How can I tell whether a certain element is contained in a list or array?

(portions of this answer contributed by Anno Siegel and brian d foy)

Hearing the word "in" is an *indication* that you probably should have used a hash, not a list or array, to store your data. Hashes are designed to answer this question quickly and efficiently. Arrays aren't.

That being said, there are several ways to approach this. In Perl 5.10 and later, you can use the smart match operator to check that an item is contained in an array or a hash:

```
use 5.010;  
  
if( $item ~~ @array ) {  
    say "The array contains $item"  
}  
  
if( $item ~~ %hash ) {  
    say "The hash contains $item"  
}
```

With earlier versions of Perl, you have to do a bit more work. If you are going to make this query many times over arbitrary string values, the fastest way is probably to invert the original array and maintain a hash whose keys are the first array's values:

```
my @blues = qw/azure cerulean teal turquoise lapis-lazuli/;  
my %is_blue = ();  
for (@blues) { $is_blue{$_} = 1 }
```

Now you can check whether `$is_blue{$some_color}`. It might have been a good idea to keep the blues all in a hash in the first place.

If the values are all small integers, you could use a simple indexed array. This kind of an array will take up less space:

```
my @primes = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31);  
my @is_tiny_prime = ();  
for (@primes) { $is_tiny_prime[$_] = 1 }  
# or simply @istiny_prime[@primes] = (1) x @primes;
```

Now you check whether `$is_tiny_prime[$some_number]`.

If the values in question are integers instead of strings, you can save quite a lot of space by using bit strings instead:

```
my @articles = ( 1..10, 150..2000, 2017 );  
undef $read;  
for (@articles) { vec($read,$_ ,1) = 1 }
```

Now check whether `vec($read,$n,1)` is true for some `$n`.

These methods guarantee fast individual tests but require a re-organization of the original list or array. They only pay off if you have to test multiple values against the same array.

If you are testing only once, the standard module *List::Util* exports the function `first` for this purpose. It works by stopping once it finds the element. It's written in C for speed, and its Perl equivalent looks like this subroutine:

```
sub first (&@) {
    my $code = shift;
    foreach (@_) {
        return $_ if &{$code}();
    }
    undef;
}
```

If speed is of little concern, the common idiom uses `grep` in scalar context (which returns the number of items that passed its condition) to traverse the entire list. This does have the benefit of telling you how many matches it found, though.

```
my $is_there = grep $_ eq $whatever, @array;
```

If you want to actually extract the matching elements, simply use `grep` in list context.

```
my @matches = grep $_ eq $whatever, @array;
```

## How do I compute the difference of two arrays? How do I compute the intersection of two arrays?

Use a hash. Here's code to do both and more. It assumes that each element is unique in a given array:

```
my (@union, @intersection, @difference);
my %count = ();
foreach my $element (@array1, @array2) { $count{$element}++ }
foreach my $element (keys %count) {
    push @union, $element;
    push @{$count{$element} > 1 ? \@intersection : \@difference },
    $element;
}
```

Note that this is the *symmetric difference*, that is, all elements in either A or in B but not in both. Think of it as an xor operation.

## How do I test whether two arrays or hashes are equal?

With Perl 5.10 and later, the smart match operator can give you the answer with the least amount of work:

```
use 5.010;

if( @array1 ~~ @array2 ) {
    say "The arrays are the same";
}

if( %hash1 ~~ %hash2 ) # doesn't check values! {
    say "The hash keys are the same";
}
```

```
}
```

The following code works for single-level arrays. It uses a stringwise comparison, and does not distinguish defined versus undefined empty strings. Modify if you have other needs.

```
$are_equal = compare_arrays(\@frogs, \@toads);

sub compare_arrays {
    my ($first, $second) = @_;
    no warnings; # silence spurious -w undef complaints
    return 0 unless @$first == @$second;
    for (my $i = 0; $i < @$first; $i++) {
        return 0 if $first->[$i] ne $second->[$i];
    }
    return 1;
}
```

For multilevel structures, you may wish to use an approach more like this one. It uses the CPAN module *FreezeThaw*:

```
use FreezeThaw qw(cmpStr);
my @a = my @b = ( "this", "that", [ "more", "stuff" ] );

printf "a and b contain %s arrays\n",
    cmpStr(\@a, \@b) == 0
    ? "the same"
    : "different";
```

This approach also works for comparing hashes. Here we'll demonstrate two different answers:

```
use FreezeThaw qw(cmpStr cmpStrHard);

my %a = my %b = ( "this" => "that", "extra" => [ "more", "stuff" ] );
$a{EXTRA} = \%b;
$b{EXTRA} = \%a;

printf "a and b contain %s hashes\n",
    cmpStr(\%a, \%b) == 0 ? "the same" : "different";

printf "a and b contain %s hashes\n",
    cmpStrHard(\%a, \%b) == 0 ? "the same" : "different";
```

The first reports that both those the hashes contain the same data, while the second reports that they do not. Which you prefer is left as an exercise to the reader.

### How do I find the first array element for which a condition is true?

To find the first array element which satisfies a condition, you can use the `first()` function in the *List::Util* module, which comes with Perl 5.8. This example finds the first element that contains "Perl".

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

my $element = first { /Perl/ } @array;
```

If you cannot use *List::Util*, you can make your own loop to do the same thing. Once you find the



element, you stop the loop with `last`.

```
my $found;
foreach ( @array ) {
    if( /Perl/ ) { $found = $_; last }
}
```

If you want the array index, use the `firstidx()` function from `List::MoreUtils`:

```
use List::MoreUtils qw(firstidx);
my $index = firstidx { /Perl/ } @array;
```

Or write it yourself, iterating through the indices and checking the array element at each index until you find one that satisfies the condition:

```
my( $found, $index ) = ( undef, -1 );
for( $i = 0; $i < @array; $i++ ) {
    if( $array[$i] =~ /Perl/ ) {
        $found = $array[$i];
        $index = $i;
        last;
    }
}
```

## How do I handle linked lists?

(contributed by brian d foy)

Perl's arrays do not have a fixed size, so you don't need linked lists if you just want to add or remove items. You can use array operations such as `push`, `pop`, `shift`, `unshift`, or `splice` to do that.

Sometimes, however, linked lists can be useful in situations where you want to "shard" an array so you have many small arrays instead of a single big array. You can keep arrays longer than Perl's largest array index, lock smaller arrays separately in threaded programs, reallocate less memory, or quickly insert elements in the middle of the chain.

Steve Lembark goes through the details in his YAPC::NA 2009 talk "Perly Linked Lists" (<http://www.slideshare.net/lembark/perly-linked-lists>), although you can just use his `LinkedList::Single` module.

## How do I handle circular lists?

(contributed by brian d foy)

If you want to cycle through an array endlessly, you can increment the index modulo the number of elements in the array:

```
my @array = qw( a b c );
my $i = 0;

while( 1 ) {
    print $array[ $i++ % @array ], "\n";
    last if $i > 20;
}
```

You can also use `Tie::Cycle` to use a scalar that always has the next element of the circular array:

```
use Tie::Cycle;
```

```
tie my $cycle, 'Tie::Cycle', [ qw( FFFFFFF 000000 FFFF00 ) ];

print $cycle; # FFFFFFF
print $cycle; # 000000
print $cycle; # FFFF00
```

The *Array::Iterator::Circular* creates an iterator object for circular arrays:

```
use Array::Iterator::Circular;

my $color_iterator = Array::Iterator::Circular->new(
    qw(red green blue orange)
);

foreach ( 1 .. 20 ) {
    print $color_iterator->next, "\n";
}
```

### How do I shuffle an array randomly?

If you either have Perl 5.8.0 or later installed, or if you have *Scalar-List-Utils* 1.03 or later installed, you can say:

```
use List::Util 'shuffle';

@shuffled = shuffle(@list);
```

If not, you can use a Fisher-Yates shuffle.

```
sub fisher_yates_shuffle {
    my $deck = shift; # $deck is a reference to an array
    return unless @$deck; # must not be empty!

    my $i = @$deck;
    while (--$i) {
        my $j = int rand ($i+1);
        @$deck[$i,$j] = @$deck[$j,$i];
    }
}

# shuffle my mpeg collection
#
my @mpeg = <audio/*/*.mp3>;
fisher_yates_shuffle( \@mpeg ); # randomize @mpeg in place
print @mpeg;
```

Note that the above implementation shuffles an array in place, unlike the `List::Util::shuffle()` which takes a list and returns a new shuffled list.

You've probably seen shuffling algorithms that work using splice, randomly picking another element to swap the current element with

```
srand;
@new = ();
@old = 1 .. 10; # just a demo
```

```
while (@old) {
    push(@new, splice(@old, rand @old, 1));
}
```

This is bad because splice is already  $O(N)$ , and since you do it  $N$  times, you just invented a quadratic algorithm; that is,  $O(N^2)$ . This does not scale, although Perl is so efficient that you probably won't notice this until you have rather largish arrays.

## How do I process/modify each element of an array?

Use for/foreach:

```
for (@lines) {
    s/foo/bar/;    # change that word
    tr/XZ/ZX/;    # swap those letters
}
```

Here's another; let's compute spherical volumes:

```
my @volumes = @radii;
for (@volumes) {    # @volumes has changed parts
    $_ **= 3;
    $_ *= (4/3) * 3.14159; # this will be constant folded
}
```

which can also be done with `map()` which is made to transform one list into another:

```
my @volumes = map {$_ ** 3 * (4/3) * 3.14159} @radii;
```

If you want to do the same thing to modify the values of the hash, you can use the `values` function. As of Perl 5.6 the values are not copied, so if you modify `$orbit` (in this case), you modify the value.

```
for my $orbit ( values %orbits ) {
    ($orbit **= 3) *= (4/3) * 3.14159;
}
```

Prior to perl 5.6 `values` returned copies of the values, so older perl code often contains constructions such as `@orbits{keys %orbits}` instead of `values %orbits` where the hash is to be modified.

## How do I select a random element from an array?

Use the `rand()` function (see "*rand*" in *perlfunc*):

```
my $index = rand @array;
my $element = $array[$index];
```

Or, simply:

```
my $element = $array[ rand @array ];
```

## How do I permute N elements of a list?

Use the *List::Permutor* module on CPAN. If the list is actually an array, try the *Algorithm::Permute* module (also on CPAN). It's written in XS code and is very efficient:

```
use Algorithm::Permute;

my @array = 'a'..'d';
```

```
my $p_iterator = Algorithm::Permute->new ( \@array );

while (my @perm = $p_iterator->next) {
    print "next permutation: (@perm)\n";
}
```

For even faster execution, you could do:

```
use Algorithm::Permute;

my @array = 'a'..'d';

Algorithm::Permute::permute {
    print "next permutation: (@array)\n";
} @array;
```

Here's a little program that generates all permutations of all the words on each line of input. The algorithm embodied in the `permute()` function is discussed in Volume 4 (still unpublished) of Knuth's *The Art of Computer Programming* and will work on any list:

```
#!/usr/bin/perl -n
# Fischer-Krause ordered permutation generator

sub permute (&@) {
    my $code = shift;
    my @idx = 0..$_;
    while ( $code->(@_[@idx]) ) {
        my $p = $#idx;
        --$p while $idx[$p-1] > $idx[$p];
        my $q = $p or return;
        push @idx, reverse splice @idx, $p;
        ++$q while $idx[$p-1] > $idx[$q];
        @idx[$p-1,$q]=@idx[$q,$p-1];
    }
}

permute { print "@_\n" } split;
```

The *Algorithm::Loops* module also provides the `NextPermute` and `NextPermuteNum` functions which efficiently find all unique permutations of an array, even if it contains duplicate values, modifying it in-place: if its elements are in reverse-sorted order then the array is reversed, making it sorted, and it returns false; otherwise the next permutation is returned.

`NextPermute` uses string order and `NextPermuteNum` numeric order, so you can enumerate all the permutations of 0..9 like this:

```
use Algorithm::Loops qw(NextPermuteNum);

my @list= 0..9;
do { print "@list\n" } while NextPermuteNum @list;
```

## How do I sort an array by (anything)?

Supply a comparison function to `sort()` (described in "sort" in *perlfunc*):

```
@list = sort { $a <=> $b } @list;
```

The default sort function is `cmp`, string comparison, which would sort `(1, 2, 10)` into `(1, 10, 2)`. `<=>`, used above, is the numerical comparison operator.

If you have a complicated function needed to pull out the part you want to sort on, then don't do it inside the sort function. Pull it out first, because the sort BLOCK can be called many times for the same element. Here's an example of how to pull out the first word after the first number on each item, and then sort those words case-insensitively.

```
my @idx;
for (@data) {
    my $item;
    ($item) = /\d+\s*(\S+)/;
    push @idx, uc($item);
}
my @sorted = @data[ sort { $idx[$a] cmp $idx[$b] } 0 .. $#idx ];
```

which could also be written this way, using a trick that's come to be known as the Schwartzian Transform:

```
my @sorted = map { $_->[0] }
    sort { $a->[1] cmp $b->[1] }
    map { [ $_, uc( /\d+\s*(\S+)/ )[0] ] } @data;
```

If you need to sort on several fields, the following paradigm is useful.

```
my @sorted = sort {
    field1($a) <=> field1($b) ||
    field2($a) cmp field2($b) ||
    field3($a) cmp field3($b)
} @data;
```

This can be conveniently combined with precalculation of keys as given above.

See the *sort* article in the "Far More Than You Ever Wanted To Know" collection in <http://www.cpan.org/misc/olddoc/FMTEYEWTK.tgz> for more about this approach.

See also the question later in *perlfaq4* on sorting hashes.

## How do I manipulate arrays of bits?

Use `pack()` and `unpack()`, or else `vec()` and the bitwise operations.

For example, you don't have to store individual bits in an array (which would mean that you're wasting a lot of space). To convert an array of bits to a string, use `vec()` to set the right bits. This sets `$vec` to have bit `N` set only if `$ints[N]` was set:

```
my @ints = (...); # array of bits, e.g. ( 1, 0, 0, 1, 1, 0 ... )
my $vec = '';
foreach( 0 .. $#ints ) {
    vec($vec,$_,1) = 1 if $ints[$_];
}
```

The string `$vec` only takes up as many bits as it needs. For instance, if you had 16 entries in `@ints`, `$vec` only needs two bytes to store them (not counting the scalar variable overhead).

Here's how, given a vector in `$vec`, you can get those bits into your `@ints` array:

```
sub bitvec_to_list {
    my $vec = shift;
    my @ints;
    # Find null-byte density then select best algorithm
    if ($vec =~ tr/\0// / length $vec > 0.95) {
        use integer;
        my $i;

        # This method is faster with mostly null-bytes
        while($vec =~ /[^\0]/g ) {
            $i = -9 + 8 * pos $vec;
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
        }
    }
    else {
        # This method is a fast general algorithm
        use integer;
        my $bits = unpack "b*", $vec;
        push @ints, 0 if $bits =~ s/^(\d)// && $1;
        push @ints, pos $bits while($bits =~ /1/g);
    }

    return \@ints;
}
```

This method gets faster the more sparse the bit vector is. (Courtesy of Tim Bunce and Winfried Koenig.)

You can make the while loop a lot shorter with this suggestion from Benjamin Goldberg:

```
while($vec =~ /[^\0]+/g ) {
    push @ints, grep vec($vec, $_, 1), $-[0] * 8 .. $+[0] * 8;
}
```

Or use the CPAN module *Bit::Vector*:

```
my $vector = Bit::Vector->new($num_of_bits);
$vector->Index_List_Store(@ints);
my @ints = $vector->Index_List_Read();
```

*Bit::Vector* provides efficient methods for bit vector, sets of small integers and "big int" math.

Here's a more extensive illustration using `vec()`:

```
# vec demo
my $vector = "\xff\x0f\xef\xfe";
print "Ilya's string \\xff\\x0f\\xef\\xfe represents the number ",
unpack("N", $vector), "\n";
my $is_set = vec($vector, 23, 1);
```

```
print "Its 23rd bit is ", $is_set ? "set" : "clear", ".\n";
pvec($vector);

set_vec(1,1,1);
set_vec(3,1,1);
set_vec(23,1,1);

set_vec(3,1,3);
set_vec(3,2,3);
set_vec(3,4,3);
set_vec(3,4,7);
set_vec(3,8,3);
set_vec(3,8,7);

set_vec(0,32,17);
set_vec(1,32,17);

sub set_vec {
    my ($offset, $width, $value) = @_;
    my $vector = '';
    vec($vector, $offset, $width) = $value;
    print "offset=$offset width=$width value=$value\n";
    pvec($vector);
}

sub pvec {
    my $vector = shift;
    my $bits = unpack("b*", $vector);
    my $i = 0;
    my $BASE = 8;

    print "vector length in bytes: ", length($vector), "\n";
    @bytes = unpack("A8" x length($vector), $bits);
    print "bits are: @bytes\n\n";
}
```

### Why does defined() return true on empty arrays and hashes?

The short story is that you should probably only use defined on scalars or functions, not on aggregates (arrays and hashes). See *"defined" in perlfunc* in the 5.004 release or later of Perl for more detail.

## Data: Hashes (Associative Arrays)

### How do I process an entire hash?

(contributed by brian d foy)

There are a couple of ways that you can process an entire hash. You can get a list of keys, then go through each key, or grab a one key-value pair at a time.

To go through all of the keys, use the `keys` function. This extracts all of the keys of the hash and gives them back to you as a list. You can then get the value through the particular key you're processing:

```
foreach my $key ( keys %hash ) {
    my $value = $hash{$key}
```

```
    ...
}
```

Once you have the list of keys, you can process that list before you process the hash elements. For instance, you can sort the keys so you can process them in lexical order:

```
foreach my $key ( sort keys %hash ) {
    my $value = $hash{$key}
    ...
}
```

Or, you might want to only process some of the items. If you only want to deal with the keys that start with `text:`, you can select just those using `grep`:

```
foreach my $key ( grep /^text:/, keys %hash ) {
    my $value = $hash{$key}
    ...
}
```

If the hash is very large, you might not want to create a long list of keys. To save some memory, you can grab one key-value pair at a time using `each()`, which returns a pair you haven't seen yet:

```
while( my( $key, $value ) = each( %hash ) ) {
    ...
}
```

The `each` operator returns the pairs in apparently random order, so if ordering matters to you, you'll have to stick with the `keys` method.

The `each()` operator can be a bit tricky though. You can't add or delete keys of the hash while you're using it without possibly skipping or re-processing some pairs after Perl internally rehashes all of the elements. Additionally, a hash has only one iterator, so if you mix `keys`, `values`, or `each` on the same hash, you risk resetting the iterator and messing up your processing. See the `each` entry in *perlfunc* for more details.

## How do I merge two hashes?

(contributed by brian d foy)

Before you decide to merge two hashes, you have to decide what to do if both hashes contain keys that are the same and if you want to leave the original hashes as they were.

If you want to preserve the original hashes, copy one hash (`%hash1`) to a new hash (`%new_hash`), then add the keys from the other hash (`%hash2`) to the new hash. Checking that the key already exists in `%new_hash` gives you a chance to decide what to do with the duplicates:

```
my %new_hash = %hash1; # make a copy; leave %hash1 alone

foreach my $key2 ( keys %hash2 ) {
    if( exists $new_hash{$key2} ) {
        warn "Key [$key2] is in both hashes!";
        # handle the duplicate (perhaps only warning)
        ...
        next;
    }
    else {
        $new_hash{$key2} = $hash2{$key2};
    }
}
```



```
}
```

If you don't want to create a new hash, you can still use this looping technique; just change the `%new_hash` to `%hash1`.

```
foreach my $key2 ( keys %hash2 ) {
    if( exists $hash1{$key2} ) {
        warn "Key [$key2] is in both hashes!";
        # handle the duplicate (perhaps only warning)
        ...
        next;
    }
    else {
        $hash1{$key2} = $hash2{$key2};
    }
}
```

If you don't care that one hash overwrites keys and values from the other, you could just use a hash slice to add one hash to another. In this case, values from `%hash2` replace values from `%hash1` when they have keys in common:

```
@hash1{ keys %hash2 } = values %hash2;
```

## What happens if I add or remove keys from a hash while iterating over it?

(contributed by brian d foy)

The easy answer is "Don't do that!"

If you iterate through the hash with `each()`, you can delete the key most recently returned without worrying about it. If you delete or add other keys, the iterator may skip or double up on them since perl may rearrange the hash table. See the entry for `each()` in *perlfunc*.

## How do I look up a hash element by value?

Create a reverse hash:

```
my %by_value = reverse %by_key;
my $key = $by_value{$value};
```

That's not particularly efficient. It would be more space-efficient to use:

```
while (my ($key, $value) = each %by_key) {
    $by_value{$value} = $key;
}
```

If your hash could have repeated values, the methods above will only find one of the associated keys. This may or may not worry you. If it does worry you, you can always reverse the hash into a hash of arrays instead:

```
while (my ($key, $value) = each %by_key) {
    push @{$key_list_by_value{$value}}, $key;
}
```

## How can I know how many entries are in a hash?

(contributed by brian d foy)

This is very similar to "How do I process an entire hash?", also in *perlfaq4*, but a bit simpler in the

common cases.

You can use the `keys()` built-in function in scalar context to find out how many entries you have in a hash:

```
my $key_count = keys %hash; # must be scalar context!
```

If you want to find out how many entries have a defined value, that's a bit different. You have to check each value. A `grep` is handy:

```
my $defined_value_count = grep { defined } values %hash;
```

You can use that same structure to count the entries any way that you like. If you want the count of the keys with vowels in them, you just test for that instead:

```
my $vowel_count = grep { /[aeiou]/ } keys %hash;
```

The `grep` in scalar context returns the count. If you want the list of matching items, just use it in list context instead:

```
my @defined_values = grep { defined } values %hash;
```

The `keys()` function also resets the iterator, which means that you may see strange results if you use this between uses of other hash operators such as `each()`.

## How do I sort a hash (optionally by value instead of key)?

(contributed by brian d foy)

To sort a hash, start with the keys. In this example, we give the list of keys to the `sort` function which then compares them ASCIIbetically (which might be affected by your locale settings). The output list has the keys in ASCIIbetical order. Once we have the keys, we can go through them to create a report which lists the keys in ASCIIbetical order.

```
my @keys = sort { $a cmp $b } keys %hash;

foreach my $key ( @keys ) {
    printf "%-20s %6d\n", $key, $hash{$key};
}
```

We could get more fancy in the `sort()` block though. Instead of comparing the keys, we can compute a value with them and use that value as the comparison.

For instance, to make our report order case-insensitive, we use `lc` to lowercase the keys before comparing them:

```
my @keys = sort { lc $a cmp lc $b } keys %hash;
```

Note: if the computation is expensive or the hash has many elements, you may want to look at the Schwartzian Transform to cache the computation results.

If we want to sort by the hash value instead, we use the hash key to look it up. We still get out a list of keys, but this time they are ordered by their value.

```
my @keys = sort { $hash{$a} <=> $hash{$b} } keys %hash;
```

From there we can get more complex. If the hash values are the same, we can provide a secondary sort on the hash key.

```
my @keys = sort {
    $hash{$a} <=> $hash{$b}
    or
    "\L$a" cmp "\L$b"
} keys %hash;
```

## How can I always keep my hash sorted?

You can look into using the `DB_File` module and `tie()` using the `$DB_BTREE` hash bindings as documented in *"In Memory Databases" in DB\_File*. The `Tie::IxHash` module from CPAN might also be instructive. Although this does keep your hash sorted, you might not like the slowdown you suffer from the tie interface. Are you sure you need to do this? :)

## What's the difference between "delete" and "undef" with hashes?

Hashes contain pairs of scalars: the first is the key, the second is the value. The key will be coerced to a string, although the value can be any kind of scalar: string, number, or reference. If a key `$key` is present in `%hash`, `exists($hash{$key})` will return true. The value for a given key can be `undef`, in which case `$hash{$key}` will be `undef` while `exists $hash{$key}` will return true. This corresponds to `($key, undef)` being in the hash.

Pictures help... Here's the `%hash` table:

keys	values
a	3
x	7
d	0
e	2

And these conditions hold

<code>\$hash{'a'}</code>	is true
<code>\$hash{'d'}</code>	is false
<code>defined \$hash{'d'}</code>	is true
<code>defined \$hash{'a'}</code>	is true
<code>exists \$hash{'a'}</code>	is true (Perl 5 only)
<code>grep (\$_ eq 'a', keys %hash)</code>	is true

If you now say

```
undef $hash{'a'}
```

your table now reads:

keys	values
a	undef
x	7
d	0
e	2

and these conditions now hold; changes in caps:

<code>\$hash{'a'}</code>	is FALSE
<code>\$hash{'d'}</code>	is false

```
defined $hash{'d'}      is true
defined $hash{'a'}      is FALSE
exists $hash{'a'}       is true (Perl 5 only)
grep ($_ eq 'a', keys %hash) is true
```

Notice the last two: you have an undef value, but a defined key!

Now, consider this:

```
delete $hash{'a'}
```

your table now reads:

keys	values
x	7
d	0
e	2

and these conditions now hold; changes in caps:

```
$hash{'a'}      is false
$hash{'d'}      is false
defined $hash{'d'} is true
defined $hash{'a'} is false
exists $hash{'a'} is FALSE (Perl 5 only)
grep ($_ eq 'a', keys %hash) is FALSE
```

See, the whole entry is gone!

### Why don't my tied hashes make the defined/exists distinction?

This depends on the tied hash's implementation of EXISTS(). For example, there isn't the concept of undef with hashes that are tied to DBM\* files. It also means that exists() and defined() do the same thing with a DBM\* file, and what they end up doing is not what they do with ordinary hashes.

### How do I reset an each() operation part-way through?

(contributed by brian d foy)

You can use the keys or values functions to reset each. To simply reset the iterator used by each without doing anything else, use one of them in void context:

```
keys %hash; # resets iterator, nothing else.
values %hash; # resets iterator, nothing else.
```

See the documentation for each in *perlfunc*.

### How can I get the unique keys from two hashes?

First you extract the keys from the hashes into lists, then solve the "removing duplicates" problem described above. For example:

```
my %seen = ();
for my $element (keys(%foo), keys(%bar)) {
    $seen{$element}++;
}
my @uniq = keys %seen;
```

Or more succinctly:

```
my @uniq = keys %{ %foo,%bar };
```

Or if you really want to save space:

```
my %seen = ();
while (defined ($key = each %foo)) {
    $seen{$key}++;
}
while (defined ($key = each %bar)) {
    $seen{$key}++;
}
my @uniq = keys %seen;
```

### How can I store a multidimensional array in a DBM file?

Either stringify the structure yourself (no fun), or else get the MLDBM (which uses Data::Dumper) module from CPAN and layer it on top of either DB\_File or GDBM\_File. You might also try DBM::Deep, but it can be a bit slow.

### How can I make my hash remember the order I put elements into it?

Use the *Tie::IxHash* from CPAN.

```
use Tie::IxHash;

tie my %myhash, 'Tie::IxHash';

for (my $i=0; $i<20; $i++) {
    $myhash{$i} = 2*$i;
}

my @keys = keys %myhash;
# @keys = (0,1,2,3,...)
```

### Why does passing a subroutine an undefined element in a hash create it?

(contributed by brian d foy)

Are you using a really old version of Perl?

Normally, accessing a hash key's value for a nonexistent key will *not* create the key.

```
my %hash = ();
my $value = $hash{ 'foo' };
print "This won't print\n" if exists $hash{ 'foo' };
```

Passing `$hash{ 'foo' }` to a subroutine used to be a special case, though. Since you could assign directly to `$_[0]`, Perl had to be ready to make that assignment so it created the hash key ahead of time:

```
my_sub( $hash{ 'foo' } );
print "This will print before 5.004\n" if exists $hash{ 'foo' };

sub my_sub {
    # $_[0] = 'bar'; # create hash key in case you do this
    1;
}
```

```
}
```

Since Perl 5.004, however, this situation is a special case and Perl creates the hash key only when you make the assignment:

```
my_sub( $hash{ 'foo' } );
print "This will print, even after 5.004\n" if exists $hash{ 'foo' };

sub my_sub {
    $_[0] = 'bar';
}
```

However, if you want the old behavior (and think carefully about that because it's a weird side effect), you can pass a hash slice instead. Perl 5.004 didn't make this a special case:

```
my_sub( @hash{ qw/foo/ } );
```

## How can I make the Perl equivalent of a C structure/C++ class/hash or array of hashes or arrays?

Usually a hash ref, perhaps like this:

```
$record = {
    NAME    => "Jason",
    EMPNO   => 132,
    TITLE   => "deputy peon",
    AGE     => 23,
    SALARY  => 37_000,
    PALS    => [ "Norbert", "Rhys", "Phineas" ],
};
```

References are documented in *perlref* and *perlreftut*. Examples of complex data structures are given in *perldsc* and *perllo!*. Examples of structures and object-oriented classes are in *perltoot*.

## How can I use a reference as a hash key?

(contributed by brian d foy and Ben Morrow)

Hash keys are strings, so you can't really use a reference as the key. When you try to do that, perl turns the reference into its stringified form (for instance, `HASH(0xDEADBEEF)`). From there you can't get back the reference from the stringified form, at least without doing some extra work on your own.

Remember that the entry in the hash will still be there even if the referenced variable goes out of scope, and that it is entirely possible for Perl to subsequently allocate a different variable at the same address. This will mean a new variable might accidentally be associated with the value for an old.

If you have Perl 5.10 or later, and you just want to store a value against the reference for lookup later, you can use the core `Hash::Util::Fieldhash` module. This will also handle renaming the keys if you use multiple threads (which causes all variables to be reallocated at new addresses, changing their stringification), and garbage-collecting the entries when the referenced variable goes out of scope.

If you actually need to be able to get a real reference back from each hash entry, you can use the `Tie::RefHash` module, which does the required work for you.

## How can I check if a key exists in a multilevel hash?

(contributed by brian d foy)

The trick to this problem is avoiding accidental autovivification. If you want to check three keys deep, you might naïvely try this:

```
my %hash;
if( exists $hash{key1}{key2}{key3} ) {
    ...;
}
```

Even though you started with a completely empty hash, after that call to `exists` you've created the structure you needed to check for `key3`:

```
%hash = (
    'key1' => {
        'key2' => {}
    }
);
```

That's autovivification. You can get around this in a few ways. The easiest way is to just turn it off. The lexical autovivification pragma is available on CPAN. Now you don't add to the hash:

```
{
    no autovivification;
    my %hash;
    if( exists $hash{key1}{key2}{key3} ) {
        ...;
    }
}
```

The *Data::Diver* module on CPAN can do it for you too. Its `Dive` subroutine can tell you not only if the keys exist but also get the value:

```
use Data::Diver qw(Dive);

my @exists = Dive( \%hash, qw(key1 key2 key3) );
if( ! @exists ) {
    ...; # keys do not exist
}
elsif( ! defined $exists[0] ) {
    ...; # keys exist but value is undef
}
```

You can easily do this yourself too by checking each level of the hash before you move onto the next level. This is essentially what *Data::Diver* does for you:

```
if( check_hash( \%hash, qw(key1 key2 key3) ) ) {
    ...;
}

sub check_hash {
    my( $hash, @keys ) = @_;

    return unless @keys;

    foreach my $key ( @keys ) {
        return unless eval { exists $hash->{$key} };
        $hash = $hash->{$key};
    }
}
```

```
    return 1;
}
```

## How can I prevent addition of unwanted keys into a hash?

Since version 5.8.0, hashes can be *restricted* to a fixed number of given keys. Methods for creating and dealing with restricted hashes are exported by the *Hash::Util* module.

## Data: Misc

### How do I handle binary data correctly?

Perl is binary-clean, so it can handle binary data just fine. On Windows or DOS, however, you have to use *binmode* for binary files to avoid conversions for line endings. In general, you should use *binmode* any time you want to work with binary data.

Also see "*binmode*" in *perlfunc* or *perlopentut*.

If you're concerned about 8-bit textual data then see *perllocale*. If you want to deal with multibyte characters, however, there are some gotchas. See the section on Regular Expressions.

### How do I determine whether a scalar is a number/whole/integer/float?

Assuming that you don't care about IEEE notations like "NaN" or "Infinity", you probably just want to use a regular expression:

```
use 5.010;

given( $number ) {
    when( /\D/ )
        { say "\thas nondigits"; continue }
    when( /^d+z/ )
        { say "\tis a whole number"; continue }
    when( /^-?d+z/ )
        { say "\tis an integer"; continue }
    when( /^[+-]?d+z/ )
        { say "\tis a +/- integer"; continue }
    when( /^-?(?:d+\.?|\.\d)\d*z/ )
        { say "\tis a real number"; continue }
    when( /^[+-]?(?=\.\d)\d*\.\d*(?:e[+-]?d+)?z/i )
        { say "\tis a C float" }
}
```

There are also some commonly used modules for the task. *Scalar::Util* (distributed with 5.8) provides access to perl's internal function *looks\_like\_number* for determining whether a variable looks like a number. *Data::Types* exports functions that validate data types using both the above and other regular expressions. Thirdly, there is *Regexp::Common* which has regular expressions to match various types of numbers. Those three modules are available from the CPAN.

If you're on a POSIX system, Perl supports the *POSIX::strtod* function for converting strings to doubles (and also *POSIX::strtol* for longs). Its semantics are somewhat cumbersome, so here's a *getnum* wrapper function for more convenient access. This function takes a string and returns the number it found, or *undef* for input that isn't a C float. The *is\_numeric* function is a front end to *getnum* if you just want to say, "Is this a float?"

```
sub getnum {
    use POSIX qw(strtod);
    my $str = shift;
    $str =~ s/^\s+//;
    $str =~ s/\s+$//;
```



```
$! = 0;
my($num, $unparsed) = strtod($str);
if (($str eq '') || ($unparsed != 0) || $!) {
    return undef;
}
else {
    return $num;
}
}

sub is_numeric { defined getnum($_[0]) }
```

Or you could check out the *String::Scanf* module on the CPAN instead.

### How do I keep persistent data across program calls?

For some specific applications, you can use one of the DBM modules. See *AnyDBM\_File*. More generically, you should consult the *FreezeThaw* or *Storable* modules from CPAN. Starting from Perl 5.8, *Storable* is part of the standard distribution. Here's one example using *Storable*'s *store* and *retrieve* functions:

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

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

### How do I print out or copy a recursive data structure?

The *Data::Dumper* module on CPAN (or the 5.005 release of Perl) is great for printing out data structures. The *Storable* module on CPAN (or the 5.8 release of Perl), provides a function called *dclone* that recursively copies its argument.

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

Where *\$r1* can be a reference to any kind of data structure you'd like. It will be deeply copied. Because *dclone* takes and returns references, you'd have to add extra punctuation if you had a hash or arrays that you wanted to copy.

```
%newhash = %{ dclone(\%oldhash) };
```

### How do I define methods for every class/object?

(contributed by Ben Morrow)

You can use the *UNIVERSAL* class (see *UNIVERSAL*). However, please be very careful to consider the consequences of doing this: adding methods to every object is very likely to have unintended consequences. If possible, it would be better to have all your object inherit from some common base class, or to use an object system like *Moose* that supports roles.

### How do I verify a credit card checksum?

Get the *Business::CreditCard* module from CPAN.

### How do I pack arrays of doubles or floats for XS code?

The *arrays.h/arrays.c* code in the *PGPLOT* module on CPAN does just this. If you're doing a lot of float or double processing, consider using the *PDL* module from CPAN instead--it makes

number-crunching easy.

See <http://search.cpan.org/dist/PGPLOT> for the code.

## AUTHOR AND COPYRIGHT

Copyright (c) 1997-2010 Tom Christiansen, Nathan Torkington, and other authors as noted. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in this file are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.