

NAME

perlfaq4 - Data Manipulation (\$Revision: 1.56 \$, \$Date: 2004/11/03 22:47:56 \$)

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)?

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 show 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 the *Floating Point Arithmetic* 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 above item "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?

Perl only understands octal and hex numbers as such when they occur as literals in your program. Octal literals in perl must start with a leading "0" and hexadecimal literals must start with a leading "0x". If they are read in from somewhere and assigned, no automatic conversion takes place. You must explicitly use `oct()` or `hex()` if you want the values converted to decimal. `oct()` interprets hex ("0x350"), octal ("0350" or even without the leading "0", like "377") and binary ("0b1010") numbers, while `hex()` only converts hexadecimal ones, with or without a leading "0x", like "0x255", "3A", "ff", or "deadbeef". The inverse mapping from decimal to octal can be done with either the "%o" or "%O" `sprintf()` formats.

This problem shows up most often when people try using `chmod()`, `mkdir()`, `umask()`, or `sysopen()`, which by widespread tradition typically take permissions in octal.

```
chmod(644, $file); # WRONG
chmod(0644, $file); # right
```

Note the mistake in the first line was specifying the decimal literal 644, rather than the intended octal literal 0644. The problem can be seen with:

```
printf("%#o",644); # prints 01204
```

Surely you had not intended `chmod(01204, $file);` - did you? If you want to use numeric literals as arguments to `chmod()` et al. then please try to express them as octal constants, that is with a leading zero and with the following digits restricted to the set 0..7.

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;
$ceil = ceil(3.5); # 4
$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 rounding is being used by Perl, but to instead implement the rounding function you need yourself.

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

```
for ($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 below 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:

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

Using the hex function:

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

Using pack:

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

Using the CPAN module `Bit::Vector`:

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

How do I convert from decimal to hexadecimal

Using `sprintf`:

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

Using `unpack`:

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

Using `Bit::Vector`:

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

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

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

How do I convert from octal to decimal

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

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

Using the `oct` function:

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

Using `Bit::Vector`:

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

How do I convert from decimal to octal

Using `sprintf`:

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

Using `Bit::Vector`:

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

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

Using oct:

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

Using pack and ord:

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

Using pack and unpack for larger strings:

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

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

Using Bit::Vector:

```
$vec = Bit::Vector->new_Bin(32,
"11011110101011011011111011101111");
$dec = $vec->to_Dec();
```

How do I convert from decimal to binary

Using sprintf (perl 5.6+):

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

Using unpack:

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

Using Bit::Vector:

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

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

For example:

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

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

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

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

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

but you should be aware that the `..` operator creates an array of all integers in the range. This can take a lot of memory for large ranges. Instead use:

```
@results = ();  
for ($i=5; $i < 500_005; $i++) {  
    push(@results, some_func($i));  
}
```

This situation has been fixed in Perl5.005. Use of `..` in a `for` loop will iterate over the range, without creating the entire range.

```
for my $i (5 .. 500_005) {  
    push(@results, some_func($i));  
}
```

will not create a 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 :-). see 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."

If you want numbers that are more random than `rand` with `srand` provides, you should also check out the `Math::TrulyRandom` module from CPAN. It uses the imperfections in your system's timer to generate random numbers, but this takes quite a while. If you want a better pseudorandom generator than comes with your operating system, look at "Numerical Recipes in C" at <http://www.nr.com/> .

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

`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 );
```

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_in(50,120)`.

```
sub random_int_in ($$) {
    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 `localtime` function returns the day of the week. Without an argument `localtime` uses the current time.

```
$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 the `Time::Local` module to get a time in epoch seconds for the argument to `localtime`.

```
use POSIX qw/strftime/;
use Time::Local;
my $week_of_year = strftime "%W",
    localtime( timelocal( 0, 0, 0, 18, 11, 1987 ) );
```

The `Date::Calc` module provides two functions for to calculate these.

```
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 reliably determine the current century or millennium.

How can I compare two dates and find the difference?

If you're storing your dates as epoch seconds then simply subtract one from the other. If you've got a structured date (distinct year, day, month, hour, minute, seconds values), then for reasons of accessibility, simplicity, and efficiency, merely use either `timelocal` or `timegm` (from the `Time::Local` module in the standard distribution) to reduce structured dates to epoch seconds. However, if you don't know the precise format of your dates, then you should probably use either of the `Date::Manip` and `Date::Calc` modules from CPAN before you go hacking up your own parsing routine to handle arbitrary date formats.

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` and `Date::Manip` modules from CPAN.

How can I find the Julian Day?

Use the `Time::JulianDay` module (part of the `Time-modules` bundle available from CPAN.)

Before you immerse yourself too deeply in this, be sure to verify that it is the *Julian* Day you really want. Are you interested in a way of getting serial days so that you just can tell how many days they are apart or so that you can do also other date arithmetic? If you are interested in performing date arithmetic, this can be done using modules `Date::Manip` or `Date::Calc`.

There is too many details and much confusion on this issue to cover in this FAQ, but the term is applied (correctly) to a calendar now supplanted by the Gregorian Calendar, with the Julian Calendar failing to adjust properly for leap years on centennial years (among other annoyances). The term is also used (incorrectly) to mean: [1] days in the Gregorian Calendar; and [2] days since a particular starting time or 'epoch', usually 1970 in the Unix world and 1980 in the MS-DOS/Windows world. If you find that it is not the first meaning that you really want, then check out the `Date::Manip` and `Date::Calc` modules. (Thanks to David Cassell for most of this text.)

How do I find yesterday's date?

If you only need to find the date (and not the same time), you can use the `Date::Calc` module.

```
use Date::Calc qw( Today Add_Delta_Days );

my @date = Add_Delta_Days( Today(), -1 );

print "@date\n";
```

Most people try to use the time rather than the calendar to figure out dates, but that assumes that your 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. Russ Allbery offers this solution.

```
sub yesterday {
    my $now = defined $_[0] ? $_[0] : time;
    my $then = $now - 60 * 60 * 24;
    my $ndst = (localtime $now)[8] > 0;
    my $stdst = (localtime $then)[8] > 0;
    $then - ($stdst - $ndst) * 60 * 60;
}
```

Should give you "this time yesterday" in seconds since epoch relative to the first argument or the current time if no argument is given and suitable for passing to localtime or whatever else you need to do with it. \$ndst is whether we're currently in daylight savings time; \$stdst is whether the point 24 hours ago was in daylight savings time. If \$stdst and \$ndst are the same, a boundary wasn't crossed, and the correction will subtract 0. If \$stdst is 1 and \$ndst is 0, subtract an hour more from yesterday's time since we gained an extra hour while going off daylight savings time. If \$stdst is 0 and \$ndst is 1, subtract a negative hour (add an hour) to yesterday's time since we lost an hour.

All of this is because during those days when one switches off or onto DST, a "day" isn't 24 hours long; it's either 23 or 25.

The explicit settings of \$ndst and \$stdst are necessary because localtime only says it returns the system tm struct, and the system tm struct at least on Solaris doesn't guarantee any particular positive value (like, say, 1) for isdst, just a positive value. And that value can potentially be negative, if DST information isn't available (this sub just treats those cases like no DST).

Note that between 2am and 3am on the day after the time zone switches off daylight savings time, the exact hour of "yesterday" corresponding to the current hour is not clearly defined. Note also that if used between 2am and 3am the day after the change to daylight savings time, the result will be between 3am and 4am of the previous day; it's arguable whether this is correct.

This sub does not attempt to deal with leap seconds (most things don't).

Does Perl have a Year 2000 problem? Is Perl Y2K compliant?

Short answer: No, Perl does not have a Year 2000 problem. Yes, Perl is Y2K compliant (whatever that means). The programmers you've hired to use it, however, probably are not.

Long answer: The question belies a true understanding of the issue. Perl is just as Y2K compliant as your pencil--no more, and no less. Can you use your pencil to write a non-Y2K-compliant memo? Of course you can. Is that the pencil's fault? Of course it isn't.

The date and time functions supplied with Perl (gmtime and localtime) supply adequate information to determine the year well beyond 2000 (2038 is when trouble strikes for 32-bit machines). The year returned by these functions when used in a list context is the year minus 1900. For years between 1910 and 1999 this *happens* to be a 2-digit decimal number. To avoid the year 2000 problem simply do not treat the year as a 2-digit number. It isn't.

When gmtime() and localtime() are used in scalar context they return a timestamp string that contains a fully-expanded year. For example, \$timestamp = gmtime(1005613200) sets \$timestamp to "Tue Nov 13 01:00:00 2001". There's no year 2000 problem here.

That doesn't mean that Perl can't be used to create non-Y2K compliant programs. It can. But so can your pencil. It's the fault of the user, not the language. At the risk of inflaming the NRA: "Perl doesn't break Y2K, people do." See <http://www.perl.org/about/y2k.html> for a longer exposition.

Data: Strings

How do I validate input?

The answer to this question is usually a regular expression, perhaps with auxiliary logic. See the more specific questions (numbers, mail addresses, etc.) for details.

How do I unescape a string?

It depends just what you mean by "unescape". 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?

To turn "abbcccd" into "abccd":

```
s/(.)\1/$1/g; # add /s to include newlines
```

Here's a solution that turns "abbcccd" to "abcd":

```
y///cs; # y == tr, but shorter :-)
```

How do I expand function calls in a string?

This is documented in *perlref*. In general, this is fraught with quoting and readability problems, but it is possible. To interpolate a subroutine call (in list context) into a string:

```
print "My sub returned @{$[mysub(1,2,3)]} that time.\n";
```

See also "How can I expand variables in text strings?" in this section of the FAQ.

How do I find matching/nesting anything?

This isn't something that can be done in one regular expression, no matter how complicated. 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. But none of these deals with nested patterns. For balanced expressions using `(`, `{`, `[` or `<` as delimiters, use the CPAN module `Regexp::Common`, or see "*??{code}*" in *perlre*. For other cases, you'll have to write a parser.

If you are serious about writing a parser, there are a number of modules or oddities that will make your life a lot easier. There are the CPAN modules `Parse::RecDescent`, `Parse::Yapp`, and `Text::Balanced`; and the `byacc` program. Starting from perl 5.8 the `Text::Balanced` is part of the standard distribution.

One simple destructive, inside-out approach that you might try is to pull out the smallest nesting parts one at a time:

```
while (s/BEGIN((?:(!BEGIN)(?!END).)*)END//gs) {  
  # do something with $1  
}
```

A more complicated and sneaky approach is to make Perl's regular expression engine do it for you. This is courtesy Dean Inada, and rather has the nature of an Obfuscated Perl Contest entry, but it really does work:

```
# $_ contains the string to parse  
# BEGIN and END are the opening and closing markers for the
```

```
# nested text.

@() = ('','');
@() = ('','');
($re=$_)=~s/((BEGIN)|(END)|.)/$[!$3]\Q$1\E$([!$2]/gs;
@$ = (eval{/$re/},$@!~/unmatched/i);
print join("\n",@$[0..$#$]) if( $$[-1] );
```

How do I reverse a string?

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

```
$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;
@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.

```
$string = "Just another Perl Hacker";
$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:

```

$string = "ThisXlineXhasXsomeXx'sXinXit";
$count = ($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:

```

$string = "-9 55 48 -2 23 -76 4 14 -44";
while ($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.

```

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

```

How do I capitalize all the words on one line?

To make the first letter of each word upper case:

```

$line =~ s/\b(\w)/\U$1/g;

```

This has the strange effect of turning "don't do it" into "Don'T Do It". Sometimes you might

want this. Other times you might need a more thorough solution (Suggested by brian d foy):

```
$string =~ s/ (
    (^\w)      #at the beginning of the line
    |         # or
    (\s\w)    #preceded by whitespace
)
/\U$1/xg;
$string =~ /([\w']+)/\u\L$1/g;
```

To make the whole line upper case:

```
$line = uc($line);
```

To force each word to be lower case, with the first letter upper case:

```
$line =~ s/(\w+)/\u\L$1/g;
```

You can (and probably should) enable locale awareness of those characters by placing a `use locale` pragma in your program. See *perllocale* for endless details on locales.

This is sometimes referred to as putting something into "title case", but that's not quite accurate. Consider the proper capitalization of the movie *Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb*, for example.

Damian Conway's *Text::Autoformat* module provides some smart case transformations:

```
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 can I split a [character] delimited string except when inside [character]?

Several modules can handle this sort of parsing---*Text::Balanced*, *Text::CVS*, *Text::CVS_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`):

```
@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);

```

There's also a `Text::CSV` (Comma-Separated Values) module on CPAN.

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

Although the simplest approach would seem to be

```
$string =~ s/^\s*(.*?)\s*$/\1/;
```

not only is this unnecessarily slow and destructive, it also fails with embedded newlines. It is much faster to do this operation in two steps:

```

$string =~ s/^\s+//;
$string =~ s/\s+$//;

```

Or more nicely written as:

```

for ($string) {
    s/^\s+//;
    s/\s+$//;
}

```

This idiom takes advantage of the `foreach` loop's aliasing behavior to factor out common code. You can do this on several strings at once, or arrays, or even the values of a hash if you use a slice:

```

# trim whitespace in the scalar, the array,
# and all the values in the hash
foreach ($scalar, @array, @hash{keys %hash}) {
    s/^\s+//;
    s/\s+$//;
}

```

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):
$padding = sprintf("%${pad_len}s", $text);
$padding = sprintf("%*s", $pad_len, $text); # same thing

```

```

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

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

# Right padding a string with blanks using pack (will truncate):
$padding = pack("A$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:

```

$padding = $pad_char x ( $pad_len - length( $text ) ) . $text;
$padding = $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?

Use `substr()` or `unpack()`, both documented in *perlfunc*. If you prefer thinking in terms of columns instead of widths, you can use this kind of thing:

```

# determine the unpack format needed to split Linux ps output
# arguments are cut columns
my $fmt = cut2fmt(8, 14, 20, 26, 30, 34, 41, 47, 59, 63, 67, 72);

sub cut2fmt {
my(@positions) = @_;
my $template = '';
my $lastpos = 1;
for my $place (@positions) {
    $template .= "A" . ($place - $lastpos) . " ";
    $lastpos = $place;
}
$template .= "A*";
return $template;
}

```

How do I find the soundex value of a string?

Use the standard `Text::Soundex` module distributed with Perl. Before you do so, you may want to determine whether ``soundex'` is in fact what you think it is. Knuth's soundex algorithm compresses words into a small space, and so it does not necessarily distinguish between two words which you might want to appear separately. For example, the last names ``Knuth'` and ``Kant'` are both mapped to the soundex code `K530`. If `Text::Soundex` does not do what you are looking for, you might want to consider the `String::Approx` module available at CPAN.

How can I expand variables in text strings?

Let's assume that you have a string like:

```
$text = 'this has a $foo in it and a $bar';
```

If those were both global variables, then this would suffice:

```
$text =~ s/\$(\w+)/${$1}/g; # no /e needed
```

But since they are probably lexicals, or at least, they could be, you'd have to do this:

```
$text =~ s/(\$\w+)/$1/eeg;
die if $@; # needed /ee, not /e
```

It's probably better in the general case to treat those variables as entries in some special hash. For example:

```
%user_defs = (
  foo => 23,
  bar => 19,
);
$text =~ s/\$(\w+)/$user_defs{$1}/g;
```

See also "How do I expand function calls in a string?" in this section of the FAQ.

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
$new = "$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;
$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.

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

Why don't my <<HERE documents work?

Check for these three things:

There must be no space after the << part.

There (probably) should be a semicolon at the end.

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

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

```
# all in one
($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.

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

This works with leading special strings, dynamically determined:

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



```

@@@     TAINT_NOT;
@@@     return 0;
@@@ }
        MAIN_INTERPRETER_LOOP

```

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

```

$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?

An array has a changeable length. A list does not. An array is something you can push or pop, while a list is a set of values. Some people make the distinction that a list is a value while an array is a variable. Subroutines are passed and return lists, you put things into list context, you initialize arrays with lists, and you `foreach()` across a list. `@` variables are arrays, anonymous arrays are arrays, arrays in scalar context behave like the number of elements in them, subroutines access their arguments through the array `@_`, and `push/pop/shift` only work on arrays.

As a side note, there's no such thing as a list in scalar context. When you say

```
$scalar = (2, 5, 7, 9);
```

you're using the comma operator in scalar context, so it uses the scalar comma operator. There never was a list there at all! This causes the last value to be returned: 9.

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

The former is a scalar value; the latter an array slice, making it a list with one (scalar) value. You should use `$` when you want a scalar value (most of the time) and `@` when you want a list with one scalar value in it (very, very rarely; nearly never, in fact).

Sometimes it doesn't make a difference, but sometimes it does. For example, compare:

```
$good[0] = 'some program that outputs several lines';
```

with

```
@bad[0] = 'same program that outputs several lines';
```

The use `warnings` pragma and the `-w` flag will warn you about these matters.

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

There are several possible ways, depending on whether the array is ordered and whether you wish to preserve the ordering.

a)

If `@in` is sorted, and you want `@out` to be sorted: (this assumes all true values in the array)

```
$prev = "not equal to $in[0]";
```

```
@out = grep($_ ne $prev && ($prev = $_, 1), @in);
```

This is nice in that it doesn't use much extra memory, simulating `uniq(1)`'s behavior of removing only adjacent duplicates. The `", 1"` guarantees that the expression is true (so that `grep` picks it up) even if the `$_` is `0`, `""`, or `undef`.

b)

If you don't know whether `@in` is sorted:

```
undef %saw;
@out = grep(!$saw{$_}++, @in);
```

c)

Like (b), but `@in` contains only small integers:

```
@out = grep(!$saw[$_]++, @in);
```

d)

A way to do (b) without any loops or greps:

```
undef %saw;
@saw{@in} = ();
@out = sort keys %saw; # remove sort if undesired
```

e)

Like (d), but `@in` contains only small positive integers:

```
undef @ary;
@ary{@in} = @in;
@out = grep {defined} @ary;
```

But perhaps you should have been using a hash all along, eh?

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

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. 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.

```
@blues = qw/azure cerulean teal turquoise lapis-lazuli/;
%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:

```
@primes = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31);
@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:

```
@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`.

Please do not use

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

or worse yet

```
($is_there) = grep /$whatever/, @array;
```

These are slow (checks every element even if the first matches), inefficient (same reason), and potentially buggy (what if there are regex characters in `$whatever`?). If you're only testing once, then use:

```
$is_there = 0;
foreach $elt (@array) {
if ($elt eq $elt_to_find) {
    $is_there = 1;
    last;
}
}
if ($is_there) { ... }
```

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:

```
@union = @intersection = @difference = ();
%count = ();
foreach $element (@array1, @array2) { $count{$element}++ }
foreach $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?

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);
@a = @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);

%a = %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 my $element ( @array )
{
    if( /Perl/ ) { $found = $element; last }
}

```

If you want the array index, you can iterate through the indices and check 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?

In general, you usually don't need a linked list in Perl, since with regular arrays, you can push and pop or shift and unshift at either end, or you can use splice to add and/or remove arbitrary number of elements at arbitrary points. Both pop and shift are both O(1) operations on Perl's dynamic arrays. In the absence of shifts and pops, push in general needs to reallocate on the order every log(N) times, and unshift will need to copy pointers each time.

If you really, really wanted, you could use structures as described in *perldsc* or *perltoot* and do just what the algorithm book tells you to do. For example, imagine a list node like this:

```
$node = {
    VALUE => 42,
    LINK  => undef,
};
```

You could walk the list this way:

```
print "List: ";
for ($node = $head; $node; $node = $node->{LINK}) {
    print $node->{VALUE}, " ";
}
print "\n";
```

You could add to the list this way:

```
my ($head, $tail);
$tail = append($head, 1);          # grow a new head
for $value ( 2 .. 10 ) {
    $tail = append($tail, $value);
}

sub append {
    my($list, $value) = @_;
    my $node = { VALUE => $value };
    if ($list) {
        $node->{LINK} = $list->{LINK};
        $list->{LINK} = $node;
    } else {
        $_[0] = $node;          # replace caller's version
    }
    return $node;
}
```

But again, Perl's built-in are virtually always good enough.

How do I handle circular lists?

Circular lists could be handled in the traditional fashion with linked lists, or you could just do something like this with an array:

```
unshift(@array, pop(@array)); # the last shall be first
push(@array, shift(@array)); # and vice versa
```

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
    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
    y/XZ/ZX/; # swap those letters
}
```

```
}

```

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

```
    for (@volumes = @radii) {    # @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:

```
@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 $orbit ( values %orbits ) {
($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*):

```
$index = rand @array;
$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-Kause 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;
```

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.

```

@idx = ();
for (@data) {
    ($item) = /\d+\s*(\S+)/;
    push @idx, uc($item);
}
@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:

```

@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.

```

@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 below on sorting hashes.

How do I manipulate arrays of bits?

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

For example, this sets `$vec` to have bit `N` set if `$ints[N]` was set:

```
$vec = '';
foreach(@ints) { vec($vec,$_,1) = 1 }
```

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`:

```
$vector = Bit::Vector->new($num_of_bits);
$vector->Index_List_Store(@ints);
@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
$vector = "\xff\x0f\xef\xfe";
print "Ilya's string \\xff\\x0f\\xef\\xfe represents the number ",
unpack("N", $vector), "\n";
$is_set = vec($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?

Use the each() function (see *"each" in perlfunc*) if you don't care whether it's sorted:

```

while ( ($key, $value) = each %hash) {
print "$key = $value\n";
}

```

If you want it sorted, you'll have to use `foreach()` on the result of sorting the keys as shown in an earlier question.

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:

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

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

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

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

If you mean how many keys, then all you have to do is use the `keys()` function in a scalar context:

```
$num_keys = keys %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)?

Internally, hashes are stored in a way that prevents you from imposing an order on key-value pairs. Instead, you have to sort a list of the keys or values:

```
@keys = sort keys %hash; # sorted by key
@keys = sort {
    $hash{$a} cmp $hash{$b}
} keys %hash; # and by value
```

Here we'll do a reverse numeric sort by value, and if two keys are identical, sort by length of key, or if that fails, by straight ASCII comparison of the keys (well, possibly modified by your locale--see *perllocale*).

```
@keys = sort {
    $hash{$b} <=> $hash{$a}
    ||

```

```
length($b) <=> length($a)
||
    $a cmp $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.

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

```

$hash{'a'}           is true
$hash{'d'}           is false
defined $hash{'d'}   is true
defined $hash{'a'}   is true
exists $hash{'a'}    is true (Perl5 only)
grep ($_ eq 'a', keys %hash) 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:

```

$hash{'a'}           is FALSE
$hash{'d'}           is false
defined $hash{'d'}   is true
defined $hash{'a'}   is FALSE
```

```
exists $hash{'a'}           is true (Perl5 only)
grep ($_ 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 $hash{'a'}    is FALSE (Perl5 only)
grep ($_ 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?

Using `keys %hash` in scalar context returns the number of keys in the hash *and* resets the iterator associated with the hash. You may need to do this if you use `last` to exit a loop early so that when you re-enter it, the hash iterator has been reset.

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:

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

Or more succinctly:

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

Or if you really want to save space:

```
%seen = ();
```

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

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?

If you say something like:

```
somefunc($hash{"nonesuch key here"});
```

Then that element "autovivifies"; that is, it springs into existence whether you store something there or not. That's because functions get scalars passed in by reference. If somefunc() modifies \$_[0], it has to be ready to write it back into the caller's version.

This has been fixed as of Perl5.004.

Normally, merely accessing a key's value for a nonexistent key does *not* cause that key to be forever there. This is different than awk's behavior.

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 the upcoming *perlreftut*. Examples of complex data structures are given in *perldsc* and *perllo1*. Examples of structures and object-oriented classes are in *perltot*.

How can I use a reference as a hash key?

You can't do this directly, but you could use the standard `Tie::RefHash` module distributed with Perl.

Data: Misc

How do I handle binary data correctly?

Perl is binary clean, so this shouldn't be a problem. For example, this works fine (assuming the files are found):

```

    if (`cat /vmunix` =~ /gzip/) {
    print "Your kernel is GNU-zip enabled!\n";
    }

```

On less elegant (read: Byzantine) systems, however, you have to play tedious games with "text" versus "binary" files. See *"binmode" in perlfunc or perlomentut*.

If you're concerned about 8-bit ASCII 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.

```

    if (/\\D/)           { print "has nondigits\n" }
    if (/^\d+$/)         { print "is a whole number\n" }
    if (/^-?\d+$/)      { print "is an integer\n" }
    if (/^[+-]?\d+$/)   { print "is a +/- integer\n" }
    if (/^-?\d+\.\d*$/) { print "is a real number\n" }
    if (/^-?(?:\d+(?:\.\d*)?|\.\d+)$/ ) { print "is a decimal number\n" }
    if (/^([+-]?)(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?$/)
    { print "a C float\n" }

```

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. 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. The POSIX module (part of the standard Perl distribution) provides the `strtod` and `strtol` for converting strings to double and longs, respectively.

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 of arrays that you wanted to copy.

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

How do I define methods for every class/object?

Use the *UNIVERSAL* class (see *UNIVERSAL*).

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 `kgbpack.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.

AUTHOR AND COPYRIGHT

Copyright (c) 1997-2002 Tom Christiansen and Nathan Torkington. 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.