Solutions & Examples for Perl Programmers

2nd Edition

Perl Cookbook

O'REILLY®

Tom Christiansen & Nathan Torkington

Perl Cookbook

Other Perl resources from O'Reilly

Related titles

Programming Perl

Learning Perl

Perl Cookbook

CGI Programming with Perl

Computer Science & Perl Programming

Embedding Perl in HTML with Mason

Mastering Regular Expressions

Perl for System Administration

Programming Web Services with Perl

Perl Pocket Reference

Perl in a Nutshell

Perl Graphics Programming

Perl Books Resource Center

perl.oreilly.com is a complete catalog of O'Reilly's books on Perl and related technologies, including sample chapters and code examples.

O'REILLY® perl.com THE SOURCE FOR PERL

Perl.com is the central web site for the Perl community. It is the perfect starting place for finding out everything there is to know about Perl.

Conferences

O'Reilly & Associates brings diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.

O'REILLY NETWORK Safari® Bookshelf™

Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today with a free trial.

SECOND EDITION

Perl Cookbook

Tom Christiansen and Nathan Torkington

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 1
Strings

He multiplieth words without knowledge.
—Job 35:16

1.0 Introduction

Many programming languages force you to work at an uncomfortably low level. You think in lines, but your language wants you to deal with pointers. You think in strings, but it wants you to deal with bytes. Such a language can drive you to distraction. Don't despair; Perl isn't a low-level language, so lines and strings are easy to handle.

Perl was *designed* for easy but powerful text manipulation. In fact, Perl can manipulate text in so many ways that they can't all be described in one chapter. Check out other chapters for recipes on text processing. In particular, see Chapters 6 and 8, which discuss interesting techniques not covered here.

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

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

A scalar value is either defined or undefined. If defined, it may hold a string, number, or reference. The only undefined value is undef. All other values are defined, even numeric 0 and the empty string. Definedness is not the same as Boolean truth, though; to check whether a value is defined, use the defined function. Boolean truth has a specialized meaning, tested with operators such as && and || or in an if or while block's test condition.

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

This is seldom an issue, since conversion is automatic when the value is used numerically. If it has never been used numerically, though, and you just test whether it's true or false, you might get an unexpected answer—Boolean tests never force any sort of conversion. Adding 0 to the variable makes Perl explicitly convert the string to a number:

```
print "Gimme a number: ";
0.00000
chomp($n = <STDIN>);  # $n now holds "0.00000";

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

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

The undef value behaves like the empty string ("") when used as a string, 0 when used as a number, and the null reference when used as a reference. But in all three possible cases, it's false. Using an undefined value where Perl expects a defined value will trigger a runtime warning message on STDERR if you've enabled warnings. Merely asking whether something is true or false demands no particular value, so this is exempt from warnings. Some operations do not trigger warnings when used on variables holding undefined values. These include the autoincrement and autodecrement operators, ++ and --, and the addition and concatenation assignment operators, += and .= ("plus-equals" and "dot-equals").

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

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

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

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

Double quotes interpolate variables (but not function calls—see Recipe 1.15 to find how to do this) and expand backslash escapes. These include "\n" (newline), "\033" (the character with octal value 33), "\cJ" (Ctrl-J), "\x1B" (the character with hex value 0x1B), and so on. The full list of these is given in the *perlop*(1) manpage and the section on "Specific Characters" in Chapter 5 of *Programming Perl*.

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

If there are no backslash escapes or variables to expand within the string, it makes no difference which flavor of quotes you use. When choosing between writing 'this' and writing "this", some Perl programmers prefer to use double quotes so that the strings stand out. This also avoids the slight risk of having single quotes mistaken for backquotes by readers of your code. It makes no difference to Perl, and it might help readers.

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

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

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

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

Here documents are a notation borrowed from the shell used to quote a large chunk of text. The text can be interpreted as single-quoted, double-quoted, or even as commands to be executed, depending on how you quote the terminating identifier. Uninterpolated here documents do not expand the three backslash sequences the way single-quoted literals normally do. Here we double-quote two lines with a here document:

```
$a = <<"EOF";
This is a multiline here document
```

```
    terminated by EOF on a line by itself
    EOF
```

Notice there's no semicolon after the terminating `EOF`. Here documents are covered in more detail in Recipe 1.16.

The Universal Character Code

As far as the computer is concerned, all data is just a series of individual numbers, each a string of bits. Even text strings are just sequences of numeric codes interpreted as characters by programs like web browsers, mailers, printing programs, and editors.

Back when memory sizes were far smaller and memory prices far more dear, programmers would go to great lengths to save memory. Strategies such as stuffing six characters into one 36-bit word or jamming three characters into one 16-bit word were common. Even today, the numeric codes used for characters usually aren't longer than 7 or 8 bits, which are the lengths you find in ASCII and Latin1, respectively.

That doesn't leave many bits per character—and thus, not many characters. Consider an image file with 8-bit color. You're limited to 256 different colors in your palette. Similarly, with characters stored as individual *octets* (an octet is an 8-bit byte), a document can usually have no more than 256 different letters, punctuation marks, and symbols in it.

ASCII, being the *American* Standard Code for Information Interchange, was of limited utility outside the United States, since it covered only the characters needed for a slightly stripped-down dialect of American English. Consequently, many countries invented their own incompatible 8-bit encodings built upon 7-bit ASCII. Conflicting schemes for assigning numeric codes to characters sprang up, all reusing the same limited range. That meant the same number could mean a different character in different systems and that the same character could have been assigned a different number in different systems.

Locales were an early attempt to address this and other language- and country-specific issues, but they didn't work out so well for character set selection. They're still reasonable for purposes unrelated to character sets, such as local preferences for monetary units, date and time formatting, and even collating sequences. But they are of far less utility for reusing the same 8-bit namespace for different character sets.

That's because if you wanted to produce a document that used Latin, Greek, and Cyrillic characters, you were in for big trouble, since the same numeric code would be a different character under each system. For example, character number 196 is a Latin capital A with a diaeresis above it in ISO 8859-1 (Latin1); under ISO 8859-7, that same numeric code represents a Greek capital delta. So a program interpreting numeric character codes in the ISO 8859-1 locale would see one character, but under the ISO 8859-7 locale, it would see something totally different.

This makes it hard to combine different character sets in the same document. Even if you did cobble something together, few programs could work with that document's text. To know what characters you had, you'd have to know what system they were in, and you couldn't easily mix systems. If you guessed wrong, you'd get a jumbled mess on your screen, or worse.

Unicode Support in Perl

Enter Unicode.

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

Mixed-language documents are now easy, whereas before they weren't even possible. You no longer have just 128 or 256 possible characters per document. With Unicode you can have tens of thousands (and more) of different characters all jumbled together in the same document without confusion.

The problem of mixing, say, an Ä with a Δ evaporates. The first character, formally named "LATIN CAPITAL LETTER A WITH DIAERESIS" under Unicode, is assigned the code point U+00C4 (that's the Unicode preferred notation). The second, a "GREEK CAPITAL LETTER DELTA", is now at code point U+0394. With different characters always assigned different code points, there's no longer any conflict.

Perl has supported Unicode since v5.6 or so, but it wasn't until the v5.8 release that Unicode support was generally considered robust and usable. This by no coincidence corresponded to the introduction of I/O layers and their support for encodings into Perl. These are discussed in more detail in Chapter 8.

All Perl's string functions and operators, including those used for pattern matching, now operate on characters instead of octets. If you ask for a string's `length`, Perl reports how many characters are in that string, not how many bytes are in it. If you extract the first three characters of a string using `substr`, the result may or may not be three bytes. You don't know, and you shouldn't care, either. One reason not to care about the particular underlying bytewise representation is that if you have to pay attention to it, you're probably looking too closely. It shouldn't matter, really—but if it does, this might mean that Perl's implementation still has a few bumps in it. We're working on that.

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

```
$char = chr(0x394);
$code = ord($char);
```

```
    printf "char %s is code %d, %#04x\n", $char, $code, $code;

    char Δ is code 916, 0x394
```

If you test the length of that string, it will say 1, because it's just one character. Notice how we said character; we didn't say anything about its length in bytes. Certainly the internal representation requires more than just 8 bits for a numeric code that big. But you the programmer are dealing with characters as abstractions, not as physical octets. Low-level details like that are best left up to Perl.

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

You have several ways to put Unicode characters into Perl literals. If you're lucky enough to have a text editor that lets you enter Unicode directly into your Perl program, you can inform Perl you've done this via the `use utf8` pragma. Another way is to use `\x` escapes in Perl interpolated strings to indicate a character by its code point in hex, as in `\xC4`. Characters with code points above 0xFF require more than two hex digits, so these must be enclosed in braces.

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

    char Ä and Δ look different\n
```

Recipe 1.5 describes how to `use charnames` to put `\N{NAME}` escapes in string literals, such as `\N{GREEK CAPITAL LETTER DELTA}`, `\N{greek:Delta}`, or even just `\N{Delta}` to indicate a Δ character.

That's enough to get started using Unicode in Perl alone, but getting Perl to interact properly with other programs requires a bit more.

Using the old single-byte encodings like ASCII or ISO 8859-*n*, when you wrote out a character whose numeric code was *NN*, a single byte with numeric code *NN* would appear. What actually appeared depended on which fonts were available, your current locale setting, and quite a few other factors. But under Unicode, this exact duplication of logical character numbers (code points) into physical bytes emitted no longer applies. Instead, they must be encoded in any of several available output formats.

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

1.1 Accessing Substrings

Problem

You want to access or modify just a portion of a string, not the whole thing. For instance, you've read a fixed-width record and want to extract individual fields.

Solution

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

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

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

The unpack function gives only read access, but is faster when you have many substrings to extract.

```
# get a 5-byte string, skip 3 bytes,
# then grab two 8-byte strings, then the rest;
# (NB: only works on ASCII data, not Unicode)
($leading, $s1, $s2, $trailing) =
    unpack("A5 x3 A8 A8 A*", $data);

# split at 5-byte boundaries
@fivers = unpack("A5" x (length($string)/5), $string);

# chop string into individual single-byte characters
@chars  = unpack("A1" x length($string), $string);
```

Discussion

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

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

```
$string = "This is what you have";
#         +012345678901234567890  Indexing forwards  (left to right)
#          109876543210987654321- Indexing backwards (right to left)
#           note that 0 means 10 or 20, etc. above

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

You can do more than just look at parts of the string with `substr`; you can actually change them. That's because `substr` is a particularly odd kind of function—an *lvaluable* one, that is, a function whose return value may be itself assigned a value. (For the record, the others are `vec`, `pos`, and `keys`. If you squint, `local`, `my`, and `our` can also be viewed as lvaluable functions.)

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

Use the `=~` operator and the `s///`, `m//`, or `tr///` operators in conjunction with `substr` to make them affect only that portion of the string.

```
# you can test substrings with =~
if (substr($string, -10) =~ /pattern/) {
            print "Pattern matches in last 10 characters\n";
}

# substitute "at" for "is", restricted to first five characters
substr($string, 0, 5) =~ s/is/at/g;
```

You can even swap values by using several `substr`s on each side of an assignment:

```
# exchange the first and last letters in a string
$a = "make a hat";
(substr($a,0,1), substr($a,-1)) =
(substr($a,-1),  substr($a,0,1));
print $a;
take a ham
```

Although `unpack` is not lvaluable, it is considerably faster than `substr` when you extract numerous values all at once. Specify a format describing the layout of the record to unpack. For positioning, use lowercase `"x"` with a count to skip forward some number of bytes, an uppercase `"X"` with a count to skip backward some number of bytes, and an `"@"` to skip to an absolute byte offset within the record. (If the data contains Unicode strings, be careful with those three: they're strictly byte-oriented, and moving around by bytes within multibyte data is perilous at best.)

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

($b, $c) = unpack("x6 A2 X5 A2", $a); # forward 6, grab 2; backward 5, grab 2
print "$b\n$c\n";
or
be
```

Sometimes you prefer to think of your data as being cut up at specific columns. For example, you might want to place cuts right before positions 8, 14, 20, 26, and 30. Those are the column numbers where each field begins. Although you could calculate that the proper unpack format is "A7 A6 A6 A6 A4 A*", this is too much mental strain for the virtuously lazy Perl programmer. Let Perl figure it out for you. Use the cut2fmt function:

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

$fmt = cut2fmt(8, 14, 20, 26, 30);
print "$fmt\n";
A7 A6 A6 A6 A4 A*
```

The powerful unpack function goes far beyond mere text processing. It's the gateway between text and binary data.

In this recipe, we've assumed that all character data is 7- or 8-bit data so that pack's byte operations work as expected.

See Also

The pack, unpack, and substr functions in *perlfunc*(1) and in Chapter 29 of *Programming Perl*; use of the *cut2fmt* subroutine in Recipe 1.24; the binary use of unpack in Recipe 8.24

1.2 Establishing a Default Value

Problem

You would like to supply a default value to a scalar variable, but only if it doesn't already have one. It often happens that you want a hardcoded default value for a variable that can be overridden from the command line or through an environment variable.

Solution

Use the `||` or `||=` operator, which work on both strings and numbers:

```
# use $b if $b is true, else $c
$a = $b || $c;

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

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

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

# the "new" defined-or operator from future perl
use v5.9;
$a = $b // $c;
```

Discussion

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

Rather than being restricted in its return values to a mere 1 or 0 as in most other languages, Perl's `||` operator has a much more interesting property: it returns its first operand (the lefthand side) if that operand is true; otherwise it returns its second operand. The `&&` operator also returns the last evaluated expression, but is less often used for this property. These operators don't care whether their operands are strings, numbers, or references—any scalar will do. They just return the first one that makes the whole expression true or false. This doesn't affect the Boolean sense of the return value, but it does make the operators' return values more useful.

This property lets you provide a default value to a variable, function, or longer expression in case the first part doesn't pan out. Here's an example of `||`, which would set $foo to be the contents of either $bar or, if $bar were false, "DEFAULT VALUE":

```
$foo = $bar || "DEFAULT VALUE";
```

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

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

We can do this without altering @ARGV:

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

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

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

We can also write this as follows, although with slightly different semantics:

```
$dir = @ARGV ? $ARGV[0] : "/tmp";
```

This checks the number of elements in @ARGV, because the first operand (here, @ARGV) is evaluated in scalar context. It's only false when there are 0 elements, in which case we use "/tmp". In all other cases (when the user gives an argument), we use the first argument.

The following line increments a value in %count, using as the key either $shell or, if $shell is false, "/bin/sh".

```
$count{ $shell || "/bin/sh" }++;
```

You may chain several alternatives together as we have in the following example. The first expression that returns a true value will be used.

```
# find the user name on Unix systems
$user = $ENV{USER}
     || $ENV{LOGNAME}
     || getlogin( )
     || (getpwuid($<))[0]
     || "Unknown uid number $<";
```

The && operator works analogously: it returns its first operand if that operand is false; otherwise, it returns the second one. Because there aren't as many interesting false values as there are true ones, this property isn't used much. One use is demonstrated in Recipes 13.12 and 14.19.

The ||= assignment operator looks odd, but it works exactly like the other binary assignment operators. For nearly all of Perl's binary operators, $VAR OP= VALUE means $VAR = $VAR OP VALUE; for example, $a += $b is the same as $a = $a + $b. So ||= is used to set a variable when that variable is itself still false. Since the || check is a simple Boolean one—testing for truth—it doesn't care about undefined values, even when warnings are enabled.

Here's an example of ||= that sets $starting_point to "Greenwich" unless it is already set. Again, we assume $starting_point won't have the value 0 or "0", or that if it does, it's okay to change it.

```
$starting_point ||= "Greenwich";
```

You can't use or in place of || in assignments, because or's precedence is too low. $a = $b or $c is equivalent to ($a = $b) or $c. This will always assign $b to $a, which is not the behavior you want.

Don't extend this curious use of || and ||= from scalars to arrays and hashes. It doesn't work, because the operators put their left operand into scalar context. Instead, you must do something like this:

```
@a = @b unless @a;          # copy only if empty
@a = @b ? @b : @c;          # assign @b if nonempty, else @c
```

Perl is someday expected to support new operators: //, //=, and err. It may already do so by the time you read this text. These defined-or operators will work just like the logical-or operators, ||, except that they will test definedness, not mere truth. That will make the following pairs equivalent:

```
$a = defined($b) ? $b : $c;
$a = $b // $c;

$x = defined($x) ? $x : $y;
$x //= $y;

defined(read(FH, $buf, $count)  or  die "read failed: $!";
read(FH, $buf, $count)          err die "read failed: $!";
```

These three operators are already present in Perl release v5.9, which being an odd-numbered release, is an experimental version and not what you want in a production environment. It is expected to be in v5.10, which will be a stable release, and will most certainly be in Perl v6, whose release date has not yet been determined.

See Also

The || operator in *perlop*(1) and Chapter 3 of *Programming Perl*; the defined and exists functions in *perlfunc*(1) and Chapter 29 of *Programming Perl*

1.3 Exchanging Values Without Using Temporary Variables

Problem

You want to exchange the values of two scalar variables, but don't want to use a temporary variable.

Solution

Use list assignment to reorder the variables.

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

Discussion

Most programming languages require an intermediate step when swapping two variables' values:

```
$temp    = $a;
$a       = $b;
$b       = $temp;
```

Not so in Perl. It tracks both sides of the assignment, guaranteeing that you don't accidentally clobber any of your values. This eliminates the temporary variable:

```
$a       = "alpha";
$b       = "omega";
($a, $b) = ($b, $a);        # the first shall be last -- and versa vice
```

You can even exchange more than two variables at once:

```
($alpha, $beta, $production) = qw(January March August);
# move beta       to alpha,
# move production to beta,
# move alpha      to production
($alpha, $beta, $production) = ($beta, $production, $alpha);
```

When this code finishes, `$alpha`, `$beta`, and `$production` have the values `"March"`, `"August"`, and `"January"`.

See Also

The section on "List value constructors" in *perldata*(1) and on "List Values and Arrays" in Chapter 2 of *Programming Perl*

1.4 Converting Between Characters and Values

Problem

You want to print the number represented by a given character, or you want to print a character given a number.

Solution

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

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

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

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

A `C*` template used with `pack` and `unpack` can quickly convert many 8-bit bytes; similarly, use `U*` for Unicode characters.

```
@bytes = unpack("C*", $string);
$string = pack("C*", @bytes);

$unistr = pack("U4",0x24b6,0x24b7,0x24b8,0x24b9);
@unichars = unpack("U*", $unistr);
```

Discussion

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

```
$value     = ord("e");    # now 101
$character = chr(101);    # now "e"
```

If you already have a character, it's really represented as a string of length one, so just print it out directly using `print` or the `%s` format in `printf` and `sprintf`. The `%c` format forces `printf` or `sprintf` to convert a number into a character; it's not used for printing a character that's already in character format (that is, a string).

```
printf("Number %d is character %c\n", 101, 101);
```

The `pack`, `unpack`, `chr`, and `ord` functions are all faster than `sprintf`. Here are `pack` and `unpack` in action:

```
@ascii_character_numbers = unpack("C*", "sample");
print "@ascii_character_numbers\n";
115 97 109 112 108 101

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

Here's how to convert from HAL to IBM:

```
$hal = "HAL";
@byte = unpack("C*", $hal);
foreach $val (@byte) {
    $val++;                 # add one to each byte value
}
$ibm = pack("C*", @byte);
print "$ibm\n";             # prints "IBM"
```

On single-byte character data, such as plain old ASCII or any of the various ISO 8859 charsets, the `ord` function returns numbers from 0 to 255. These correspond to C's `unsigned char` data type.

However, Perl understands more than that: it also has integrated support for Unicode, the universal character encoding. If you pass `chr`, `sprintf "%c"`, or `pack "U*"` numeric values greater than 255, the return result will be a Unicode string.

Here are similar operations with Unicode:

```
@unicode_points = unpack("U*", "fac\x{0327}ade");
print "@unicode_points\n";
102 97 99 807 97 100 101

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

If all you're doing is printing out the characters' values, you probably don't even need to use `unpack`. Perl's `printf` and `sprintf` functions understand a `v` modifier that works like this:

```
printf "%vd\n", "fac\x{0327}ade";
102.97.99.807.97.100.101

printf "%vx\n", "fac\x{0327}ade";
66.61.63.327.61.64.65
```

The numeric value of each character (that is, its "code point" in Unicode parlance) in the string is emitted with a dot separator.

See Also

The `chr`, `ord`, `printf`, `sprintf`, `pack`, and `unpack` functions in *perlfunc*(1) and Chapter 29 of *Programming Perl*

1.5 Using Named Unicode Characters

Problem

You want to use Unicode names for fancy characters in your code without worrying about their code points.

Solution

Place a `use charnames` at the top of your file, then freely insert `"\N{`*`CHARSPEC`*`}"` escapes into your string literals.

Discussion

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

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

Δ is called delta.

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

Δ is an upper-case delta.

use charnames qw(cyrillic greek);
print "\N{Sigma} and \N{sigma} are Greek sigmas.\n";
print "\N{Be} and \N{be} are Cyrillic bes.\n";

Σ and σ are Greek sigmas.
Б and б are Cyrillic bes.
```

Two functions, charnames::viacode and charnames::vianame, can translate between numeric code points and the long names. The Unicode documents use the notation U+*XXXX* to indicate the Unicode character whose code point is *XXXX*, so we'll use that here in our output.

```
use charnames qw(:full);
for $code (0xC4, 0x394) {
    printf "Character U+%04X (%s) is named %s\n",
        $code, chr($code), charnames::viacode($code);
}

Character U+00C4 (Ä) is named LATIN CAPITAL LETTER A WITH DIAERESIS
Character U+0394 (Δ) is named GREEK CAPITAL LETTER DELTA

use charnames qw(:full);
$name = "MUSIC SHARP SIGN";
$code = charnames::vianame($name);
printf "%s is character U+%04X (%s)\n",
    $name, $code, chr($code);

MUSIC SHARP SIGN is character U+266F ( ♯ )
```

Here's how to find the path to Perl's copy of the Unicode character database:

```
% perl -MConfig -le 'print "$Config{privlib}/unicore/NamesList.txt"'
/usr/local/lib/perl5/5.8.1/unicore/NamesList.txt
```

Read this file to learn the character names available to you.

See Also

The *charnames*(3) manpage and Chapter 31 of *Programming Perl*; the Unicode Character Database at *http://www.unicode.org/*

1.6 Processing a String One Character at a Time

Problem

You want to process a string one character at a time.

Solution

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

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

Or extract each character in turn with a loop:

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

Discussion

As we said before, Perl's fundamental unit is the string, not the character. Needing to process anything a character at a time is rare. Usually some kind of higher-level Perl operation, like pattern matching, solves the problem more handily. See, for example, Recipe 7.14, where a set of substitutions is used to find command-line arguments.

Splitting on a pattern that matches the empty string returns a list of individual characters in the string. This is a convenient feature when done intentionally, but it's easy to do unintentionally. For instance, `/X*/` matches all possible strings, including the empty string. Odds are you will find others when you don't mean to.

Here's an example that prints the characters used in the string `"an apple a day"`, sorted in ascending order:

```
%seen = ( );
$string = "an apple a day";
foreach $char (split //, $string) {
    $seen{$char}++;
}
print "unique chars are: ", sort(keys %seen), "\n";
unique chars are:  adelnpy
```

These split and unpack solutions give an array of characters to work with. If you don't want an array, use a pattern match with the /g flag in a while loop, extracting one character at a time:

```
%seen = ();
$string = "an apple a day";
while ($string =~ /(.)/g) {
    $seen{$1}++;
}
print "unique chars are: ", sort(keys %seen), "\n";
unique chars are:  adelnpy
```

In general, whenever you find yourself doing character-by-character processing, there's probably a better way to go about it. Instead of using index and substr or split and unpack, it might be easier to use a pattern. Instead of computing a 32-bit checksum by hand, as in the next example, the unpack function can compute it far more efficiently.

The following example calculates the checksum of $string with a foreach loop. There are better checksums; this just happens to be the basis of a traditional and computationally easy checksum. You can use the standard[*] Digest::MD5 module if you want a more robust checksum.

```
$sum = 0;
foreach $byteval (unpack("C*", $string)) {
    $sum += $byteval;
}
print "sum is $sum\n";
# prints "1248" if $string was "an apple a day"
```

This does the same thing, but much faster:

```
$sum = unpack("%32C*", $string);
```

This emulates the SysV checksum program:

```
#!/usr/bin/perl
# sum - compute 16-bit checksum of all input files
$checksum = 0;
while (<>) { $checksum += unpack("%16C*", $_) }
$checksum %= (2 ** 16) - 1;
print "$checksum\n";
```

Here's an example of its use:

```
% perl sum /etc/termcap
1510
```

If you have the GNU version of *sum*, you'll need to call it with the **--sysv** option to get the same answer on the same file.

```
% sum --sysv /etc/termcap
1510 851 /etc/termcap
```

[*] It's standard as of the v5.8 release of Perl; otherwise, grab it from CPAN.

Another tiny program that processes its input one character at a time is *slowcat*, shown in Example 1-1. The idea here is to pause after each character is printed so you can scroll text before an audience slowly enough that they can read it.

Example 1-1. slowcat

```
#!/usr/bin/perl
# slowcat - emulate a   s l o w   line printer
# usage: slowcat [-DELAY] [files ...]
$DELAY = ($ARGV[0] =~ /^-([.\d]+)/) ? (shift, $1) : 1;
$| = 1;
while (<>) {
    for (split(//)) {
        print;
        select(undef,undef,undef, 0.005 * $DELAY);
    }
}
```

See Also

The `split` and `unpack` functions in *perlfunc*(1) and Chapter 29 of *Programming Perl*; the use of expanding `select` for timing is explained in Recipe 3.10

1.7 Reversing a String by Word or Character

Problem

You want to reverse the words or characters of a string.

Solution

Use the `reverse` function in scalar context for flipping characters:

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

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

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

Discussion

The `reverse` function is two different functions in one. Called in scalar context, it joins together its arguments and returns that string in reverse order. Called in list context, it returns its arguments in the opposite order. When using `reverse` for its character-flipping behavior, use `scalar` to force scalar context unless it's entirely obvious.

```
$gnirts   = reverse($string);       # reverse letters in $string

@sdrow    = reverse(@words);        # reverse elements in @words

$confused = reverse(@words);        # reverse letters in join("", @words)
```

Here's an example of reversing words in a string. Using a single space, `" "`, as the pattern to `split` is a special case. It causes `split` to use contiguous whitespace as the separator and also discard leading null fields, just like *awk*. Normally, `split` discards only trailing null fields.

```
# reverse word order
$string = 'Yoda said, "can you see this?"';
@allwords    = split(" ", $string);
$revwords    = join(" ", reverse @allwords);
print $revwords, "\n";
this?" see you "can said, Yoda
```

We could remove the temporary array `@allwords` and do it on one line:

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

Multiple whitespace in `$string` becomes a single space in `$revwords`. If you want to preserve whitespace, use this:

```
$revwords = join("", reverse split(/(\s+)/, $string));
```

One use of `reverse` is to test whether a word is a palindrome (a word that reads the same backward or forward):

```
$word = "reviver";
$is_palindrome = ($word eq reverse($word));
```

We can turn this into a one-liner that finds big palindromes in */usr/dict/words*:

```
% perl -nle 'print if $_ eq reverse && length > 5' /usr/dict/words
deedeed
degged
deified
denned
hallah
kakkak
murdrum
redder
repaper
retter
reviver
rotator
sooloos
tebbet
terret
tut-tut
```

See Also

The `split`, `reverse`, and `scalar` functions in *perlfunc*(1) and Chapter 29 of *Programming Perl*; Recipe 1.8

1.8 Treating Unicode Combined Characters as Single Characters

Problem

You have a Unicode string that contains combining characters, and you'd like to treat each of these sequences as a single logical character.

Solution

Process them using `\X` in a regular expression.

```
$string = "fac\x{0327}ade";         # "façade"
$string =~ /fa.ade/;                # fails
$string =~ /fa\Xade/;               # succeeds

@chars = split(//, $string);        # 7 letters in @chars
@chars = $string =~ /(.)/g;         # same thing
@chars = $string =~ /(\X)/g;        # 6 "letters" in @chars
```

Discussion

In Unicode, you can combine a base character with one or more non-spacing characters following it; these are usually diacritics, such as accent marks, cedillas, and tildas. Due to the presence of precombined characters, for the most part to accommodate legacy character systems, there can be two or more ways of writing the same thing.

For example, the word "façade" can be written with one character between the two a's, `"\x{E7}"`, a character right out of Latin1 (ISO 8859-1). These characters might be encoded into a two-byte sequence under the UTF-8 encoding that Perl uses internally, but those two bytes still only count as one single character. That works just fine.

There's a thornier issue. Another way to write U+00E7 is with two different code points: a regular "c" followed by `"\x{0327}"`. Code point U+0327 is a non-spacing combining character that means to go back and put a cedilla underneath the preceding base character.

There are times when you want Perl to treat each combined character sequence as one logical character. But because they're distinct code points, Perl's character-related operations treat non-spacing combining characters as separate characters, including `substr`, `length`, and regular expression metacharacters, such as in `/./` or `/[^abc]/`.

In a regular expression, the `\X` metacharacter matches an extended Unicode combining character sequence, and is exactly equivalent to `(?:\PM\pM*)` or, in long-hand:

```
(?x:            # begin non-capturing group
        \PM     # one character without the M (mark) property,
                   #   such as a letter
        \pM        # one character that does have the M (mark) property,
                   #   such as an accent mark
        *          # and you can have as many marks as you want
    )
```

Otherwise simple operations become tricky if these beasties are in your string. Consider the approaches for reversing a word by character from the previous recipe. Written with combining characters, "année" and "niño" can be expressed in Perl as "anne\x{301}e" and "nin\x{303}o".

```
    for $word ("anne\x{301}e", "nin\x{303}o") {
        printf "%s simple reversed to %s\n", $word,
            scalar reverse $word;
        printf "%s better reversed to %s\n", $word,
            join("", reverse $word =~ /\X/g);
    }
```

That produces:

```
    année simple reversed to éenna
    année better reversed to eénna
    niño simple reversed to õnin
    niño better reversed to oñin
```

In the reversals marked as simply reversed, the diacritical marking jumped from one base character to the other one. That's because a combining character always follows its base character, and you've reversed the whole string. By grabbing entire sequences of a base character plus any combining characters that follow, then reversing that list, this problem is avoided.

See Also

The *perlre*(1) and *perluniintro*(1) manpages; Chapter 15 of *Programming Perl*; Recipe 1.9

1.9 Canonicalizing Strings with Unicode Combined Characters

Problem

You have two strings that look the same when you print them out, but they don't test as string equal and sometimes even have different lengths. How can you get Perl to consider them the same strings?

Solution

When you have otherwise equivalent strings, at least some of which contain Unicode combining character sequences, instead of comparing them directly, compare the results of running them through the NFD( ) function from the Unicode::Normalize module.

```
use Unicode::Normalize;
$s1 = "fa\x{E7}ade";
$s2 = "fac\x{0327}ade";
if (NFD($s1) eq NFD($s2)) { print "Yup!\n" }
```

Discussion

The same character sequence can sometimes be specified in multiple ways. Sometimes this is because of legacy encodings, such as the letters from Latin1 that contain diacritical marks. These can be specified directly with a single character (like U+00E7, LATIN SMALL LETTER C WITH CEDILLA) or indirectly via the base character (like U+0063, LATIN SMALL LETTER C) followed by a combining character (U+0327, COMBINING CEDILLA).

Another possibility is that you have two or more marks following a base character, but the order of those marks varies in your data. Imagine you wanted the letter “c” to have both a cedilla and a caron on top of it in order to print a ç̌. That could be specified in any of these ways:

```
$string = v231.780;
#   LATIN SMALL LETTER C WITH CEDILLA
#   COMBINING CARON

$string = v99.807.780;
#         LATIN SMALL LETTER C
#         COMBINING CARON
#         COMBINING CEDILLA

$string = v99.780.807
#         LATIN SMALL LETTER C
#         COMBINING CEDILLA
#         COMBINING CARON
```

The normalization functions rearrange those into a reliable ordering. Several are provided, including NFD( ) for canonical decomposition and NFC( ) for canonical decomposition followed by canonical composition. No matter which of these three ways you used to specify your ç̌, the NFD version is v99.807.780, whereas the NFC version is v231.780.

Sometimes you may prefer NFKD( ) and NFKC( ), which are like the previous two functions except that they perform *compatible* decomposition, which for NFKC( ) is then followed by canonical composition. For example, \x{FB00} is the double-f ligature. Its NFD and NFC forms are the same thing, "\x{FB00}", but its NFKD and NFKC forms return a two-character string, "\x{66}\x{66}".

See Also

The section on "The Universal Character Code" at the beginning of this chapter; the documentation for the Unicode::Normalize module; Recipe 8.20

1.10 Treating a Unicode String as Octets

Problem

You have a Unicode string but want Perl to treat it as octets (e.g., to calculate its length or for purposes of I/O).

Solution

The `use bytes` pragma makes all Perl operations in its lexical scope treat the string as a group of octets. Use it when your code is calling Perl's character-aware functions directly:

```
$ff = "\x{FB00}";            # ff ligature
$chars = length($ff);        # length is one character
{
  use bytes;                 # force byte semantics
  $octets = length($ff);     # length is two octets
}
$chars = length($ff);        # back to character semantics
```

Alternatively, the `Encode` module lets you convert a Unicode string to a string of octets, and back again. Use it when the character-aware code isn't in your lexical scope:

```
use Encode qw(encode_utf8);

sub somefunc;                # defined elsewhere

$ff = "\x{FB00}";            # ff ligature
$ff_oct = encode_utf8($ff);  # convert to octets

$chars = somefunc($ff);      # work with character string
$octets = somefunc($ff_oct); # work with octet string
```

Discussion

As explained in this chapter's Introduction, Perl knows about two types of string: those made of simple uninterpreted octets, and those made of Unicode characters whose UTF-8 representation may require more than one octet. Each individual string has a flag associated with it, identifying the string as either UTF-8 or octets. Perl's I/O and string operations (such as `length`) check this flag and give character or octet semantics accordingly.

Sometimes you need to work with bytes and not characters. For example, many protocols have a Content-Length header that specifies the size of the body of a message in octets. You can't simply use Perl's length function to calculate the size, because if the string you're calling length on is marked as UTF-8, you'll get the size in characters.

The use bytes pragma makes all Perl functions in its lexical scope use octet semantics for strings instead of character semantics. Under this pragma, length always returns the number of octets, and read always reports the number of octets read. However, because the use bytes pragma is lexically scoped, you can't use it to change the behavior of code in another scope (e.g., someone else's function).

For this you need to create an octet-encoded copy of the UTF-8 string. In memory, of course, the same byte sequence is used for both strings. The difference is that the copy of your UTF-8 string has the UTF-8 flag cleared. Functions acting on the octet copy will give octet semantics, regardless of the scope they're in.

There is also a no bytes pragma, which forces character semantics, and a decode_utf8 function, which turns octet-encoded strings into UTF-8 encoded strings. However, these functions are less useful because not all octet strings are valid UTF-8 strings, whereas all UTF-8 strings are valid octet strings.

See Also

The documentation for the bytes pragma; the documentation for the standard Encode module

1.11 Expanding and Compressing Tabs

Problem

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

Solution

Either use a rather funny looking substitution:

```
while ($string =~ s/\t+/' ' x (length($&) * 8 - length($`) % 8)/e) {
    # spin in empty loop until substitution finally fails
}
```

or use the standard Text::Tabs module:

```
use Text::Tabs;
@expanded_lines  = expand(@lines_with_tabs);
@tabulated_lines = unexpand(@lines_without_tabs);
```

Discussion

Assuming tab stops are set every *N* positions (where *N* is customarily eight), it's easy to convert them into spaces. The standard textbook method does not use the Text::Tabs module but suffers slightly from being difficult to understand. Also, it uses the $` variable, whose very mention currently slows down every pattern match in the program. This is explained in "Special Variables" in Chapter 6. You could use this algorithm to make a filter to expand its input's tabstops to eight spaces each:

```
while (<>) {
    1 while s/\t+/' ' x (length($&) * 8 - length($`) % 8)/e;
    print;
}
```

To avoid $`, you could use a slightly more complicated alternative that uses the numbered variables for explicit capture; this one expands tabstops to four each instead of eight:

```
1 while s/^(.*?)(\t+)/$1 . ' ' x (length($2) * 4 - length($1) % 4)/e;
```

Another approach is to use the offsets directly from the @+ and @- arrays. This also expands to four-space positions:

```
1 while s/\t+/' ' x (($+[0] - $-[0]) * 4 - $-[0] % 4)/e;
```

If you're looking at all of these 1 while loops and wondering why they couldn't have been written as part of a simple s///g instead, it's because you need to recalculate the length from the start of the line again each time rather than merely from where the last match occurred.

The convention 1 while CONDITION is the same as while (CONDITION) { }, but shorter. Its origins date to when Perl ran the first incredibly faster than the second. While the second is now almost as fast, it remains convenient, and the habit has stuck.

The standard Text::Tabs module provides conversion functions to convert both directions, exports a $tabstop variable to control the number of spaces per tab, and does not incur the performance hit because it uses $1 and $2 rather than $& and $`.

```
use Text::Tabs;
$tabstop = 4;
while (<>) { print expand($_) }
```

We can also use Text::Tabs to "unexpand" the tabs. This example uses the default $tabstop value of 8:

```
use Text::Tabs;
while (<>) { print unexpand($_) }
```

See Also

The manpage for the Text::Tabs module; the s/// operator in *perlre*(1) and *perlop*(1); the @- and @+ variables (@LAST_MATCH_START and @LAST_MATCH_END) in

Chapter 28 of *Programming Perl*; the section on "When a global substitution just isn't global enough" in Chapter 5 of *Programming Perl*

1.12 Expanding Variables in User Input

Problem

You've read a string with an embedded variable reference, such as:

```
You owe $debt to me.
```

Now you want to replace `$debt` in the string with its value.

Solution

Use a substitution with symbolic references if the variables are all globals:

```
$text =~ s/\$(\w+)/${$1}/g;
```

But use a double `/ee` if they might be lexical (`my`) variables:

```
$text =~ s/(\$\w+)/$1/gee;
```

Discussion

The first technique is basically to find what looks like a variable name, then use symbolic dereferencing to interpolate its contents. If `$1` contains the string `somevar`, `${$1}` will be whatever `$somevar` contains. This won't work if the `use strict 'refs'` pragma is in effect because that bans symbolic dereferencing.

Here's an example:

```
our ($rows, $cols);
no strict 'refs';                  # for ${$1}/g below
my $text;

($rows, $cols) = (24, 80);
$text = q(I am $rows high and $cols long);  # like single quotes!
$text =~ s/\$(\w+)/${$1}/g;
print $text;
I am 24 high and 80 long
```

You may have seen the `/e` substitution modifier used to evaluate the replacement as code rather than as a string. It's designed for situations where you don't know the exact replacement value, but you do know how to calculate it. For example, doubling every whole number in a string:

```
$text = "I am 17 years old";
$text =~ s/(\d+)/2 * $1/eg;
```

When Perl is compiling your program and sees a `/e` on a substitute, it compiles the code in the replacement block along with the rest of your program, long before the substitution actually happens. When a substitution is made, $1 is replaced with the string that matched. The code to evaluate would then be something like:

```
2 * 17
```

If we tried saying:

```
$text = 'I am $AGE years old';      # note single quotes
$text =~ s/(\$\w+)/$1/eg;           # WRONG
```

assuming $text held a mention of the variable $AGE, Perl would dutifully replace $1 with $AGE and then evaluate code that looked like:

```
'$AGE'
```

which just yields us our original string back again. We need to evaluate the result *again* to get the value of the variable. To do that, just add another /e:

```
$text =~ s/(\$\w+)/$1/eeg;          # finds my() variables
```

Yes, you can have as many /e modifiers as you'd like. Only the first one is compiled and syntax-checked with the rest of your program. This makes it work like the eval {BLOCK} construct, except that it doesn't trap exceptions. Think of it more as a do {BLOCK} instead.

Subsequent /e modifiers are quite different. They're more like the eval "STRING" construct. They don't get compiled until runtime. A small advantage of this scheme is that it doesn't require a no strict 'refs' pragma for the block. A tremendous advantage is that unlike symbolic dereferencing, this mechanism finds lexical variables created with my, something symbolic references can never do.

The following example uses the /x modifier to enable whitespace and comments in the pattern part of the substitute and /e to evaluate the righthand side as code. The /e modifier gives more control over what happens in case of error or other extenuating circumstances, as we have here:

```
# expand variables in $text, but put an error message in
# if the variable isn't defined
$text =~ s{
     \$                         # find a literal dollar sign
    (\w+)                       # find a "word" and store it in $1
}{
    no strict 'refs';           # for $$1 below
    if (defined ${$1}) {
        ${$1};                    # expand global variables only
    } else {
        "[NO VARIABLE: \$$1]";  # error msg
    }
}egx;
```

Once upon a time, long ago and far away, $$1 used to mean ${$}1 when it occurred within a string; that is, the $$ variable followed by a 1. This was grandfathered to work that way so you could more readily expand the $$ variable as your process ID to compose temporary filenames. It now always means ${$1}, i.e., dereference the contents of the $1 variable. We have written it the more explicit way for clarity, not correctness.

See Also

The s/// operator in *perlre*(1) and *perlop*(1) and Chapter 5 of *Programming Perl*; the eval function in *perlfunc*(1) and Chapter 29 of *Programming Perl*; the similar use of substitutions in Recipe 20.9

1.13 Controlling Case

Problem

A string in uppercase needs converting to lowercase, or vice versa.

Solution

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

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

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

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

Discussion

The functions and string escapes look different, but both do the same thing. You can set the case of either just the first character or the whole string. You can even do both at once to force uppercase (actually, titlecase; see later explanation) on initial characters and lowercase on the rest.

```
$beast   = "dromedary";
# capitalize various parts of $beast
$capit   = ucfirst($beast);         # Dromedary
$capit   = "\u\L$beast";            # (same)
$capall  = uc($beast);              # DROMEDARY
$capall  = "\U$beast";              # (same)
$caprest = lcfirst(uc($beast));     # dROMEDARY
$caprest = "\l\U$beast";            # (same)
```

These capitalization-changing escapes are commonly used to make a string's case consistent:

```
# titlecase each word's first character, lowercase the rest
$text = "thIS is a loNG liNE";
$text =~ s/(\w+)/\u\L$1/g;
print $text;
This Is A Long Line
```

You can also use these for case-insensitive comparison:

```
if (uc($a) eq uc($b)) { # or "\U$a" eq "\U$b"
    print "a and b are the same\n";
}
```

The *randcap* program, shown in Example 1-2, randomly titlecases 20 percent of the letters of its input. This lets you converse with 14-year-old WaREz d00Dz.

Example 1-2. randcap

```
#!/usr/bin/perl -p
# randcap: filter to randomly capitalize 20% of the letters
# call to srand( ) is unnecessary as of v5.4
BEGIN { srand(time( ) ^ ($$ + ($$<<15))) }
sub randcase { rand(100) < 20 ? "\u$_[0]" : "\l$_[0]" }
s/(\w)/randcase($1)/ge;
% randcap < genesis | head -9
boOk 01 genesis
001:001 in the BEginning goD created the heaven and tHe earTh.

001:002 and the earth wAS without ForM, aND void; AnD darkneSS was
        upon The Face of the dEEp. and the spIrit of GOd movEd upOn
        tHe face of the Waters.
001:003 and god Said, let there be ligHt: and therE wAs LigHt.
```

In languages whose writing systems distinguish between uppercase and titlecase, the `ucfirst( )` function (and `\u`, its string escape alias) converts to *titlecase*. For example, in Hungarian the “dz” sequence occurs. In uppercase, it’s written as “DZ”, in titlecase as “Dz”, and in lowercase as “dz”. Unicode consequently has three different characters defined for these three situations:

```
Code point  Written   Meaning
01F1        DZ        LATIN CAPITAL LETTER DZ
01F2        Dz        LATIN CAPITAL LETTER D WITH SMALL LETTER Z
01F3        dz        LATIN SMALL LETTER DZ
```

It is tempting but ill-advised to just use `tr[a-z][A-Z]` or the like to convert case. This is a mistake because it omits all characters with diacritical markings—such as diaereses, cedillas, and accent marks—which are used in dozens of languages, including English. However, correctly handling case mappings on data with diacritical markings can be far trickier than it seems. There is no simple answer, although if everything is in Unicode, it’s not all that bad, because Perl’s case-mapping functions do work perfectly fine on Unicode data. See the section on “Universal Character Code” in the Introduction to this chapter for more information.

See Also

The `uc`, `lc`, `ucfirst`, and `lcfirst` functions in *perlfunc*(1) and Chapter 29 of *Programming Perl*; `\L`, `\U`, `\l`, and `\u` string escapes in the “Quote and Quote-like Operators” section of *perlop*(1) and Chapter 5 of *Programming Perl*

1.14 Properly Capitalizing a Title or Headline

Problem

You have a string representing a headline, the title of book, or some other work that needs proper capitalization.

Solution

Use a variant of this tc( ) titlecasing function:

```
INIT {
    our %nocap;
    for (qw(
            a an the
            and but or
            as at but by for from in into of off on onto per to with
        ))
    {
        $nocap{$_}++;
    }
}

sub tc {
    local $_ = shift;

    # put into lowercase if on stop list, else titlecase
    s/(\pL[\pL']*)/$nocap{$1} ? lc($1) : ucfirst(lc($1))/ge;

    s/^(\pL[\pL']*) /\u\L$1/x;  # last  word guaranteed to cap
    s/ (\pL[\pL']*)$/\u\L$1/x;  # first word guaranteed to cap

    # treat parenthesized portion as a complete title
    s/\( (\pL[\pL']*) /(\u\L$1/x;
    s/(\pL[\pL']*) \) /\u\L$1)/x;

    # capitalize first word following colon or semi-colon
    s/ ( [:;] \s+ ) (\pL[\pL']* ) /$1\u\L$2/x;

    return $_;
}
```

Discussion

The rules for correctly capitalizing a headline or title in English are more complex than simply capitalizing the first letter of each word. If that's all you need to do, something like this should suffice:

```
s/(\w+\S*\w*)/\u\L$1/g;
```

Most style guides tell you that the first and last words in the title should always be capitalized, along with every other word that's not an article, the particle "to" in an infinitive construct, a coordinating conjunction, or a preposition.

Here's a demo, this time demonstrating the distinguishing property of titlecase. Assume the `tc` function is as defined in the Solution.

```
# with apologies (or kudos) to Stephen Brust, PJF,
# and to JRRT, as always.
@data = (
            "the enchantress of \x{01F3}ur mountain",
    "meeting the enchantress of \x{01F3}ur mountain",
    "the lord of the rings: the fellowship of the ring",
);

$mask = "%-20s: %s\n";

sub tc_lame {
    local $_ = shift;
    s/(\w+\S*\w*)/\u\L$1/g;
    return $_;
}

for $datum (@data) {
    printf $mask, "ALL CAPITALS",       uc($datum);
    printf $mask, "no capitals",        lc($datum);
    printf $mask, "simple titlecase",   tc_lame($datum);
    printf $mask, "better titlecase",   tc($datum);
    print "\n";
}

ALL CAPITALS        : THE ENCHANTRESS OF DZUR MOUNTAIN
no capitals         : the enchantress of dzur mountain
simple titlecase    : The Enchantress Of Dzur Mountain
better titlecase    : The Enchantress of Dzur Mountain

ALL CAPITALS        : MEETING THE ENCHANTRESS OF DZUR MOUNTAIN
no capitals         : meeting the enchantress of dzur mountain
simple titlecase    : Meeting The Enchantress Of Dzur Mountain
better titlecase    : Meeting the Enchantress of Dzur Mountain

ALL CAPITALS        : THE LORD OF THE RINGS: THE FELLOWSHIP OF THE RING
no capitals         : the lord of the rings: the fellowship of the ring
simple titlecase    : The Lord Of The Rings: The Fellowship Of The Ring
better titlecase    : The Lord of the Rings: The Fellowship of the Ring
```

One thing to consider is that some style guides prefer capitalizing only prepositions that are longer than three, four, or sometimes five letters. O'Reilly & Associates, for example, keeps prepositions of four or fewer letters in lowercase. Here's a longer list of prepositions if you prefer, which you can modify to your needs:

```
@all_prepositions = qw{
    about above absent across after against along amid amidst
    among amongst around as at athwart before behind below
    beneath beside besides between betwixt beyond but by circa
```

```
        down during ere except for from in into near of off on onto
        out over past per since than through till to toward towards
        under until unto up upon versus via with within without
    };
```

This kind of approach can take you only so far, though, because it doesn't distinguish between words that can be several parts of speech. Some prepositions on the list might also double as words that should always be capitalized, such as subordinating conjunctions, adverbs, or even adjectives. For example, it's "Down by the Riverside" but "Getting By on Just $30 a Day", or "A Ringing in My Ears" but "Bringing In the Sheaves".

Another consideration is that you might prefer to apply the `\u` or `ucfirst` conversion by itself without also putting the whole string into lowercase. That way a word that's already in all capital letters, such as an acronym, doesn't lose that trait. You probably wouldn't want to convert "FBI" and "LBJ" into "Fbi" and "Lbj".

See Also

The `uc`, `lc`, `ucfirst`, and `lcfirst` functions in *perlfunc*(1) and Chapter 29 of *Programming Perl*; the `\L`, `\U`, `\l`, and `\u` string escapes in the "Quote and Quote-like Operators" section of *perlop*(1) and Chapter 5 of *Programming Perl*

1.15 Interpolating Functions and Expressions Within Strings

Problem

You want a function call or expression to expand within a string. This lets you construct more complex templates than with simple scalar variable interpolation.

Solution

Break up your expression into distinct concatenated pieces:

```
    $answer = $var1 . func() . $var2;   # scalar only
```

Or use the slightly sneaky `@{[ LIST EXPR ]}` or `${ \(SCALAR EXPR ) }` expansions:

```
    $answer = "STRING @{[ LIST EXPR ]} MORE STRING";
    $answer = "STRING ${\( SCALAR EXPR )} MORE STRING";
```

Discussion

This code shows both techniques. The first line shows concatenation; the second shows the expansion trick:

```
    $phrase = "I have " . ($n + 1) . " guanacos.";
    $phrase = "I have ${\($n + 1)} guanacos.";
```

The first technique builds the final string by concatenating smaller strings, avoiding interpolation but achieving the same end. Because print effectively concatenates its entire argument list, if we were going to print $phrase, we could have just said:

```
print "I have ",  $n + 1, " guanacos.\n";
```

When you absolutely must have interpolation, you need the punctuation-riddled interpolation from the Solution. Only @, $, and \ are special within double quotes and most backquotes. (As with m// and s///, the qx() synonym is not subject to double-quote expansion if its delimiter is single quotes! $home = qx'echo home is $HOME'; would get the shell $HOME variable, not one in Perl.) So, the only way to force arbitrary expressions to expand is by expanding a ${ } or @{ } whose block contains a reference.

In the example:

```
$phrase = "I have ${\( count_em() )} guanacos.";
```

the function call within the parentheses is not in scalar context; it is still in list context. The following overrules that:

```
$phrase = "I have ${\( scalar count_em() )} guanacos.";
```

You can do more than simply assign to a variable after interpolation. It’s a general mechanism that can be used in any double-quoted string. For instance, this example builds a string with an interpolated expression and passes the result to a function:

```
some_func("What you want is @{[ split /:/, $rec ]} items");
```

You can interpolate into a here document, as by:

```
die "Couldn't send mail" unless send_mail(<<"EOTEXT", $target);
To: $naughty
From: Your Bank
Cc: @{ get_manager_list($naughty) }
Date: @{[ do { my $now = `date`; chomp $now; $now } ]} (today)

Dear $naughty,

Today, you bounced check number @{[ 500 + int rand(100) ]} to us.
Your account is now closed.

Sincerely,
the management
EOTEXT
```

Expanding backquotes (``) is particularly challenging because you would normally end up with spurious newlines. By creating a braced block following the @ within the @{[ ]} anonymous array dereference, as in the last example, you can create private variables.

Although these techniques work, simply breaking your work up into several steps or storing everything in temporary variables is almost always clearer to the reader.

The Interpolation module from CPAN provides a more syntactically palatable covering. For example, to make elements of the hash %E evaluate and return its subscript:

```
use Interpolation E     => 'eval';
print "You bounced check number $E{500 + int rand(100)}\n";
```

Or to make a hash named %money call a suitably defined function of your choice:

```
use Interpolation money => \&currency_commify;

print "That will be $money{ 4 * $payment }, right now.\n";
```

expect to get something like:

```
That will be $3,232.421.04, right now.
```

See Also

perlref(1) and the "Other Tricks You Can Do with Hard References" section in Chapter 8 of *Programming Perl*; the Interpolation CPAN module

1.16 Indenting Here Documents

Problem

When using the multiline quoting mechanism called a *here document*, the text must be flush against the margin, which looks out of place in the code. You would like to indent the here document text in the code, but not have the indentation appear in the final string value.

Solution

Use a s/// operator to strip out leading whitespace.

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

# or with two steps
$var = << HERE_TARGET;
    your text
    goes here
HERE_TARGET
$var =~ s/^\s+//gm;
```

Discussion

The substitution is straightforward. It removes leading whitespace from the text of the here document. The /m modifier lets the ^ character match at the start of each line in the string, and the /g modifier makes the pattern-matching engine repeat the substitution as often as it can (i.e., for every line in the here document).

```
($definition = << 'FINIS') =~ s/^\s+//gm;
    The five varieties of camelids
    are the familiar camel, his friends
    the llama and the alpaca, and the
    rather less well-known guanaco
    and vicuña.
FINIS
```

Be warned: all patterns in this recipe use \s, meaning one whitespace character, which will also match newlines. This means they will remove any blank lines in your here document. If you don't want this, replace \s with [^\S\n] in the patterns.

The substitution uses the property that the result of an assignment can be used as the lefthand side of =~. This lets us do it all in one line, but works only when assigning to a variable. When you're using the here document directly, it would be considered a constant value, and you wouldn't be able to modify it. In fact, you can't change a here document's value *unless* you first put it into a variable.

Not to worry, though, because there's an easy way around this, particularly if you're going to do this a lot in the program. Just write a subroutine:

```
sub fix {
    my $string = shift;
    $string =~ s/^\s+//gm;
    return $string;
}

print fix( << "END");
    My stuff goes here
END

# With function predeclaration, you can omit the parens:
print fix << "END";
    My stuff goes here
END
```

As with all here documents, you have to place this here document's target (the token that marks its end, END in this case) flush against the lefthand margin. To have the target indented also, you'll have to put the same amount of whitespace in the quoted string as you use to indent the token.

```
($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--/;      #move attribution to line of its own
```

If you're doing this to strings that contain code you're building up for an eval, or just text to print out, you might not want to blindly strip all leading whitespace, because that would destroy your indentation. Although eval wouldn't care, your reader might.

Another embellishment is to use a special leading string for code that stands out. For example, here we'll prepend each line with @@@, properly indented:

```
if ($REMEMBER_THE_MAIN) {
    $perl_main_C = dequote << '    MAIN_INTERPRETER_LOOP';
        @@@ int
        @@@ runops( ) {
        @@@     SAVEI32(runlevel);
        @@@     runlevel++;
        @@@     while ( op = (*op->op_ppaddr)( ) ) ;
        @@@     TAINT_NOT;
        @@@     return 0;
        @@@ }
    MAIN_INTERPRETER_LOOP
    # add more code here if you want
}
```

Destroying indentation also gets you in trouble with poets.

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

Here is its sample output:

```
Here's your poem:

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
```

The following dequote function handles all these cases. It expects to be called with a here document as its argument. It checks whether each line begins with a common substring, and if so, strips that off. Otherwise, it takes the amount of leading whitespace found on the first line and removes that much from each subsequent line.

```
sub dequote {
    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 $_;
}
```

If that pattern makes your eyes glaze over, you could always break it up and add comments by adding /x:

```
if (m{
            ^                       # start of line
            \s *                    # 0 or more whitespace chars
            (?:                     # begin first non-remembered grouping
                 (                  #   begin save buffer $1
                    [^\w\s]         #     one character neither space nor word
                    +               #     1 or more of such
                 )                  #   end save buffer $1
                 ( \s* )            #   put 0 or more white in buffer $2
                 .* \n              #   match through the end of first line
             )                      # end of first grouping
             (?:                    # begin second non-remembered grouping
                \s *                #   0 or more whitespace chars
                \1                  #   whatever string is destined for $1
                \2 ?                #   what'll be in $2, but optionally
                .* \n               #   match through the end of the line
             ) +                    # now repeat that group idea 1 or more
             $                      # until the end of the line
          }x
       )
    {
        ($white, $leader) = ($2, quotemeta($1));
    } else {
        ($white, $leader) = (/^(\s+)/, '');
    }
    s{
         ^                          # start of each line (due to /m)
         \s *                       # any amount of leading whitespace
            ?                       #   but minimally matched
         $leader                    # our quoted, saved per-line leader
         (?:                        # begin unremembered grouping
           $white                   #    the same amount
         ) ?                        # optionalize in case EOL after leader
    }{ }xgm;
```

There, isn't that much easier to read? Well, maybe not; sometimes it doesn't help to pepper your code with insipid comments that mirror the code. This may be one of those cases.

See Also

The "Scalar Value Constructors" section of *perldata*(1) and the section on "Here Documents" in Chapter 2 of *Programming Perl*; the s/// operator in *perlre*(1) and *perlop*(1), and the "Pattern Matching" section in Chapter 5 of *Programming Perl*

1.17 Reformatting Paragraphs

Problem

Your string is too big to fit the screen, and you want to break it up into lines of words, without splitting a word between lines. For instance, a style correction script might read a text file a paragraph at a time, replacing bad phrases with good ones. Replacing a phrase like *utilizes the inherent functionality of* with *uses* will change the length of lines, so it must somehow reformat the paragraphs when they're output.

Solution

Use the standard Text::Wrap module to put line breaks at the right place:

```
use Text::Wrap;
@output = wrap($leadtab, $nexttab, @para);
```

Or use the more discerning CPAN module, Text::Autoformat, instead:

```
use Text::Autoformat;
$formatted = autoformat $rawtext;
```

Discussion

The Text::Wrap module provides the `wrap` function, shown in Example 1-3, which takes a list of lines and reformats them into a paragraph with no line more than `$Text::Wrap::columns` characters long. We set `$columns` to 20, ensuring that no line will be longer than 20 characters. We pass `wrap` two arguments before the list of lines: the first is the indent for the first line of output, the second the indent for every subsequent line.

Example 1-3. wrapdemo

```
#!/usr/bin/perl -w
# wrapdemo - show how Text::Wrap works
@input = ("Folding and splicing is the work of an editor,",
          "not a mere collection of silicon",
          "and",
          "mobile electrons!");
use Text::Wrap qw($columns &wrap);
$columns = 20;
print "0123456789" x 2, "\n";
print wrap("    ", "  ", @input), "\n";
```

The result of this program is:

```
01234567890123456789
    Folding and
  splicing is the
  work of an
  editor, not a
```

```
    mere collection
    of silicon and
    mobile electrons!
```

We get back a single string, with newlines ending each line but the last:

```
# merge multiple lines into one, then wrap one long line
use Text::Wrap;
undef $/;
print wrap('', '', split(/\s*\n\s*/, <>));
```

If you have the Term::ReadKey module (available from CPAN) on your system, you can determine your window size so you can wrap lines to fit the current screen size. If you don't have the module, sometimes the screen size can be found in `$ENV{COLUMNS}` or by parsing the output of the *stty*(1) command.

The following program tries to reformat both short and long lines within a paragraph, similar to the *fmt*(1) program, by setting the input record separator `$/` to the empty string (causing `<>` to read paragraphs) and the output record separator `$\` to two newlines. Then the paragraph is converted into one long line by changing all newlines and any surrounding whitespace to single spaces. Finally, we call the `wrap` function with leading and subsequent tab strings set to the empty string so we can have block paragraphs.

```
use Text::Wrap      qw(&wrap $columns);
use Term::ReadKey   qw(GetTerminalSize);
($columns) = GetTerminalSize( );
($/, $\)  = ('', "\n\n");   # read by paragraph, output 2 newlines
while (<>) {                # grab a full paragraph
    s/\s*\n\s*/ /g;         # convert intervening newlines to spaces
    print wrap('', '', $_); # and format
}
```

The CPAN module Text::Autoformat is much more clever. For one thing, it tries to avoid "widows," that is, very short lines at the end. More remarkably, it correctly copes with reformatting paragraphs that have multiple, deeply nested citations. An example from that module's manpage shows how the module can painlessly convert:

```
In comp.lang.perl.misc you wrote:
: > <CN = Clooless Noobie> writes:
: > CN> PERL sux because:
: > CN>    * It doesn't have a switch statement and you have to put $
: > CN>signs in front of everything
: > CN>    * There are too many OR operators: having |, || and 'or'
: > CN>operators is confusing
: > CN>    * VB rools, yeah!!!!!!!!!
: > CN> So anyway, how can I stop reloads on a web page?
: > CN> Email replies only, thanks - I don't read this newsgroup.
: >
: > Begone, sirrah! You are a pathetic, Bill-loving, microcephalic
: > script-infant.
: Sheesh, what's with this group - ask a question, get toasted! And how
: *dare* you accuse me of Ianuphilia!
```

into:

```
In comp.lang.perl.misc you wrote:
: > <CN = Clooless Noobie> writes:
: > CN> PERL sux because:
: > CN>    * It doesn't have a switch statement and you
: > CN>      have to put $ signs in front of everything
: > CN>    * There are too many OR operators: having |, ||
: > CN>      and 'or' operators is confusing
: > CN>    * VB rools, yeah!!!!!!!!! So anyway, how can I
: > CN>      stop reloads on a web page? Email replies
: > CN>      only, thanks - I don't read this newsgroup.
: >
: > Begone, sirrah! You are a pathetic, Bill-loving,
: > microcephalic script-infant.
: Sheesh, what's with this group - ask a question, get toasted!
: And how *dare* you accuse me of Ianuphilia!
```

simply via `print autoformat($badparagraph)`. Pretty impressive, eh?

Here's a miniprogram that uses that module to reformat each paragraph of its input stream:

```
use Text::Autoformat;
$/ = '';
while (<>) {
    print autoformat($_, {squeeze => 0, all => 1}), "\n";
}
```

See Also

The `split` and `join` functions in *perlfunc*(1) and Chapter 29 of *Programming Perl*; the manpage for the standard Text::Wrap module; the CPAN module Term::ReadKey, and its use in Recipe 15.6 and the CPAN module Text::Autoformat

1.18 Escaping Characters

Problem

You need to output a string with certain characters (quotes, commas, etc.) escaped. For instance, you're producing a format string for `sprintf` and want to convert literal `%` signs into `%%`.

Solution

Use a substitution to backslash or double each character to be escaped:

```
# backslash
$var =~ s/([CHARLIST])/\\$1/g;

# double
$var =~ s/([CHARLIST])/$1$1/g;
```

Discussion

`$var` is the variable to be altered. The `CHARLIST` is a list of characters to escape and can contain backslash escapes like `\t` and `\n`. If you just have one character to escape, omit the brackets:

```
$string =~ s/%/%%/g;
```

The following code lets you do escaping when preparing strings to submit to the shell. (In practice, you would need to escape more than just `'` and `"` to make any arbitrary string safe for the shell. Getting the list of characters right is so hard, and the risks if you get it wrong are so great, that you're better off using the list form of `system` and `exec` to run programs, shown in Recipe 16.2. They avoid the shell altogether.)

```
$string = q(Mom said, "Don't do that.");
$string =~ s/(['"])/\\$1/g;
```

We had to use two backslashes in the replacement because the replacement section of a substitution is read as a double-quoted string, and to get one backslash, you need to write two. Here's a similar example for VMS DCL, where you need to double every quote to get one through:

```
$string = q(Mom said, "Don't do that.");
$string =~ s/(['"])/$1$1/g;
```

Microsoft command interpreters are harder to work with. In Windows, *COMMAND.COM* recognizes double quotes but not single ones, disregards backquotes for running commands, and requires a backslash to make a double quote into a literal. Any of the many free or commercial Unix-like shell environments available for Windows will work just fine, though.

Because we're using character classes in the regular expressions, we can use `-` to define a range and `^` at the start to negate. This escapes all characters that aren't in the range A through Z.

```
$string =~ s/([^A-Z])/\\$1/g;
```

In practice, you wouldn't want to do that, since it would pick up a lowercase `"a"` and turn it into `"\a"`, for example, which is ASCII BEL character. (Usually when you mean non-alphabetic characters, `\PL` works better.)

If you want to escape all non-word characters, use the `\Q` and `\E` string metacharacters or the `quotemeta` function. For example, these are equivalent:

```
$string = "this \Qis a test!\E";
$string = "this is\\ a\\ test\\!";
$string = "this " . quotemeta("is a test!");
```

See Also

The `s///` operator in *perlre*(1) and *perlop*(1) and Chapter 5 of *Programming Perl*; the `quotemeta` function in *perlfunc*(1) and Chapter 29 of *Programming Perl*; the discussion of HTML escaping in Recipe 19.1; Recipe 19.5 for how to avoid having to escape strings to give the shell

1.19 Trimming Blanks from the Ends of a String

Problem

You have read a string that may have leading or trailing whitespace, and you want to remove it.

Solution

Use a pair of pattern substitutions to get rid of them:

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

Or write a function that returns the new value:

```
$string = trim($string);
@many   = trim(@many);

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

Discussion

This problem has various solutions, but this one is the most efficient for the common case. This function returns new versions of the strings passed in to it with their leading and trailing whitespace removed. It works on both single strings and lists.

To remove the last character from the string, use the chop function. Be careful not to confuse this with the similar but different chomp function, which removes the last part of the string contained within that variable if and only if it is contained in the $/ variable, "\n" by default. These are often used to remove the trailing newline from input:

```
# print what's typed, but surrounded by > < symbols
while (<STDIN>) {
    chomp;
    print ">$_<\n";
}
```

This function can be embellished in any of several ways.

First, what should you do if several strings are passed in, but the return context demands a single scalar? As written, the function given in the Solution does a somewhat silly thing: it (inadvertently) returns a scalar representing the number of strings passed in. This isn't very useful. You *could* issue a warning or raise an exception. You could also squash the list of return values together.

For strings with spans of extra whitespace at points other than their ends, you could have your function collapse any remaining stretch of whitespace characters in the interior of the string down to a single space each by adding this line as the new last line of the loop:

```
s/\s+/ /g;              # finally, collapse middle
```

That way a string like `" but\t\tnot here\n"` would become `"but not here"`. A more efficient alternative to the three substitution lines:

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

would be:

```
$_ = join(' ', split(' '));
```

If the function isn't passed any arguments at all, it could act like `chop` and `chomp` by defaulting to `$_`. Incorporating all of these embellishments produces this function:

```
# 1. trim leading and trailing white space
# 2. collapse internal whitespace to single space each
# 3. take input from $_ if no arguments given
# 4. join return list into single scalar with intervening spaces
#     if return is scalar context

sub trim {
    my @out = @_ ? @_ : $_;
    $_ = join(' ', split(' ')) for @out;
    return wantarray ? @out : "@out";
}
```

See Also

The `s///` operator in *perlre*(1) and *perlop*(1) and Chapter 5 of *Programming Perl*; the `chomp` and `chop` functions in *perlfunc*(1) and Chapter 29 of *Programming Perl*; we trim leading and trailing whitespace in the `getnum` function in Recipe 2.1

1.20 Parsing Comma-Separated Data

Problem

You have a data file containing comma-separated values that you need to read, but these data fields may have quoted commas or escaped quotes in them. Most spreadsheets and database programs use comma-separated values as a common interchange format.

Solution

If your data file follows normal Unix quoting and escaping conventions, where quotes within a field are backslash-escaped `"like \"this\""`, use the standard Text::ParseWords and this simple code:

```
use Text::ParseWords;
sub parse_csv0 {
    return quotewords("," => 0, $_[0]);
}
```

However, if quotes within a field are doubled `"like ""this"""`, you could use the following procedure from *Mastering Regular Expressions*, Second Edition:

```
sub parse_csv1 {
    my $text = shift;      # record containing comma-separated values
    my @fields  = ();

    while ($text =~ m{
        # Either some non-quote/non-comma text:
        ( [^"',] + )

         # ...or...
         |

        # ...a double-quoted field: (with "" allowed inside)

        " # field's opening quote; don't save this
         (   now a field is either
          (?:     [^"]    # non-quotes or
              |
                  ""      # adjacent quote pairs
           ) *  # any number
         )
        " # field's closing quote; unsaved

    }gx)
    {
      if (defined $1) {
          $field = $1;
      } else {
          ($field = $2) =~ s/""/"/g;
      }
      push @fields, $field;
    }
    return @fields;
}
```

Or use the CPAN Text:CSV module:

```
use Text::CSV;
sub parse_csv1 {
    my $line = shift;
    my $csv = Text::CSV->new( );
    return $csv->parse($line) && $csv->fields( );
}
```

Or use the CPAN Tie::CSV_File module:

```
    tie @data, "Tie::CSV_File", "data.csv";

    for ($i = 0; $i < @data; $i++) {
        printf "Row %d (Line %d) is %s\n", $i, $i+1, "@{$data[$i]}";
        for ($j = 0; $j < @{$data[$i]}; $j++) {
            print "Column $j is <$data[$i][$j]>\n";
        }
    }
```

Discussion

Comma-separated input is a deceptive and complex format. It sounds simple, but involves a fairly complex escaping system because the fields themselves can contain commas. This makes the pattern-matching solution complex and rules out a simple `split /,/`. Still worse, quoting and escaping conventions vary between Unix-style files and legacy systems. This incompatibility renders impossible any single algorithm for all CSV data files.

The standard Text::ParseWords module is designed to handle data whose quoting and escaping conventions follow those found in most Unix data files. This makes it eminently suitable for parsing the numerous colon-separated data files found on Unix systems, including *disktab*(5), *gettytab*(5), *printcap*(5), and *termcap*(5). Pass that module's `quotewords` function two arguments and the CSV string. The first argument is the separator (here a comma, but often a colon), and the second is a true or false value controlling whether the strings are returned with quotes around them.

In this style of data file, you represent quotation marks inside a field delimited by quotation marks by escaping them with backslashes `"like\"this\""`. Quotation marks and backslashes are the only characters that have meaning when backslashed. Any other use of a backslash will be left in the output string. The standard Text::ParseWords module's `quotewords( )` function can handle such data.

However, it's of no use on data files from legacy systems that represent quotation marks inside such a field by doubling them `"like""this"""`. For those, you'll need one of the other solutions. The first of these is based on the regular expression from *Mastering Regular Expressions,* Second Edition, by Jeffrey E. F. Friedl (O'Reilly). It enjoys the advantage of working on any system without requiring installation of modules not found in the standard distribution. In fact, it doesn't use any modules at all. Its slight disadvantage is the risk of sending the unseasoned reader into punctuation shock, despite its copious commenting.

The object-oriented CPAN module Text::CSV demonstrated in the next solution hides that parsing complexity in more easily digestible wrappers. An even more elegant solution is offered by the Tie::CSV_File module from CPAN, in which you are given what appears to be a two-dimensional array. The first dimension represents each line of the file, and the second dimension each column on each row.

Here's how you'd use our two kinds of `parse_csv` subroutines. The `q()` is just a fancy quote so we didn't have to backslash everything.

```
$line = q(XYZZY,"","O'Reilly, Inc","Wall, Larry","a \"glug\" bit,",5,"Error, Core
Dumped");
@fields = parse_csv0($line);
for ($i = 0; $i < @fields; $i++) {
    print "$i : $fields[$i]\n";
}

0 : XYZZY
1 :
2 : O'Reilly, Inc
3 : Wall, Larry
4 : a "glug" bit,
5 : 5
6 : Error, Core Dumped
```

If the second argument to `quotewords` had been 1 instead of 0, the quotes would have been retained, producing this output instead:

```
0 : XYZZY
1 : ""
2 : "O'Reilly, Inc"
3 : "Wall, Larry"
4 : "a \"glug\" bit,"
5 : 5
6 : "Error, Core Dumped"
```

The other sort of data file is manipulated the same way, but using our `parse_csv1` function instead of `parse_csv0`. Notice how the embedded quotes are doubled, not escaped.

```
$line = q(Ten Thousand,10000, 2710 ,,"10,000","It's ""10 Grand"", baby",10K);
@fields = parse_csv1($line);
for ($i = 0; $i < @fields; $i++) {
    print "$i : $fields[$i]\n";
}

0 : Ten Thousand
1 : 10000
2 :  2710
3 :
4 : 10,000
5 : It's "10 Grand", baby
6 : 10K
```

See Also

The explanation of regular expression syntax in *perlre*(1) and Chapter 5 of *Programming Perl*; the documentation for the standard Text::ParseWords module; the section on "Parsing CSV Files" in Chapter 5 of *Mastering Regular Expressions*, Second Edition

1.21 Constant Variables

Problem

You want a variable whose value cannot be modified once set.

Solution

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

```
use constant AVOGADRO => 6.02252e23;

printf "You need %g of those for guac\n", AVOGADRO;
```

If it does have to be a variable, assign to the typeglob a reference to a literal string or number, then use the scalar variable:

```
*AVOGADRO = \6.02252e23;
print "You need $AVOGADRO of those for guac\n";
```

But the most foolproof way is via a small tie class whose STORE method raises an exception:

```
package Tie::Constvar;
use Carp;
sub TIESCALAR {
    my ($class, $initval) = @_;
    my $var = $initval;
    return bless \$var => $class;
}
sub FETCH {
    my $selfref = shift;
    return $$selfref;
}
sub STORE {
    confess "Meddle not with the constants of the universe";
}
```

Discussion

The use constant pragma is the easiest to use, but has a few drawbacks. The biggest one is that it doesn't give you a variable that you can expand in double-quoted strings. Another is that it isn't scoped; it puts a subroutine of that name into the package namespace.

The way the pragma really works is to create a subroutine of that name that takes no arguments and always returns the same value (or values if a list is provided). That means it goes into the current package's namespace and isn't scoped. You could do the same thing yourself this way:

```
sub AVOGADRO( ) { 6.02252e23 }
```

If you wanted it scoped to the current block, you could make a temporary subroutine by assigning an anonymous subroutine to the typeglob of that name:

```
use subs qw(AVOGADRO);
local *AVOGADRO = sub ( ) { 6.02252e23 };
```

But that's pretty magical, so you should comment the code if you don't plan to use the pragma.

If instead of assigning to the typeglob a reference to a subroutine, you assign to it a reference to a constant scalar, then you'll be able to use the variable of that name. That's the second technique given in the Solution. Its disadvantage is that typeglobs are available only for package variables, not for lexicals created via `my`. Under the recommended `use strict` pragma, an undeclared package variable will get you into trouble, too, but you can declare the variable using `our`:

```
our $AVOGADRO;
local *AVOGADRO = \6.02252e23;
```

The third solution provided, that of creating your own little `tie` class, might appear the most complicated, but it provides the most flexibility. Plus you get to declare it as a lexical if you want.

```
tie my $AVOGADRO, Tie::Constvar, 6.02252e23;
```

After which this is okay:

```
print "You need $AVOGADRO of those for guac\n";
```

But this will get you in trouble:

```
$AVOGADRO = 6.6256e-34;   # sorry, Max
```

See Also

Recipe 1.15; Recipe 5.3; the discussion on folding constant subroutines toward the end of the section on "Compiling Your Code" in Chapter 18 of *Programming Perl*; the CPAN module Tie::Scalar::RestrictUpdates might give you some other ideas

1.22 Soundex Matching

Problem

You have two English surnames and want to know whether they sound somewhat similar, regardless of spelling. This would let you offer users a “fuzzy search” of names in a telephone book to catch “Smith” and “Smythe” and others within the set, such as “Smite” and “Smote”.

Solution

Use the standard Text::Soundex module:

```
use Text::Soundex;
$CODE  = soundex($STRING);
@CODES = soundex(@LIST);
```

Or use the CPAN module Text::Metaphone:

```
use Text::Metaphone;
$phoned_words = Metaphone('Schwern');
```

Discussion

The soundex algorithm hashes words (particularly English surnames) into a small space using a simple model that approximates an English speaker’s pronunciation of the words. Roughly speaking, each word is reduced to a four-character string. The first character is an uppercase letter; the remaining three are digits. By comparing the soundex values of two strings, we can guess whether they sound similar.

The following program prompts for a name and looks for similarly sounding names from the password file. This same approach works on any database with names, so you could key the database on the soundex values if you wanted to. Such a key wouldn’t be unique, of course.

```
use Text::Soundex;
use User::pwent;

print "Lookup user: ";
chomp($user =<STDIN>);
exit unless defined $user;
$name_code = soundex($user);

while ($uent = getpwent( )) {
    ($firstname, $lastname) = $uent->gecos =~ /(\w+)[^,]*\b(\w+)/;

    if ($name_code eq soundex($uent->name) ||
        $name_code eq soundex($lastname)   ||
        $name_code eq soundex($firstname)  )
    {
        printf "%s: %s %s\n", $uent->name, $firstname, $lastname;
    }
}
```

The Text::Metaphone module from CPAN addresses the same problem in a different and better way. The soundex function returns a letter and a three-digit code that maps just the beginning of the input string, whereas Metaphone returns a code as letters of variable length. For example:

```
                         soundex  metaphone

        Christiansen         C623     KRSXNSN
        Kris Jenson          K625     KRSJNSN

        Kyrie Eleison        K642     KRLSN
        Curious Liaison      C624     KRSLSN
```

To get the most of Metaphone, you should also use the String::Approx module from CPAN, described more fully in Recipe 6.13. It allows for there to be errors in the match and still be successful. The *edit distance* is the number of changes needed to go from one string to the next. This matches a pair of strings with an edit distance of two:

```
    if (amatch("string1", [2], "string2") { }
```

There's also an adist function that reports the edit distance. The edit distance between "Kris Jenson" "Christiansen" is 6, but between their Metaphone encodings is only 1. Likewise, the distance between the other pair is 8 originally, but down to 1 again if you compare Metaphone encodings.

```
    use Text::Metaphone qw(Metaphone);
    use String::Approx  qw(amatch);

    if (amatch(Metaphone($s1), [1], Metaphone($s1)) {
        print "Close enough!\n";
    }
```

This would successfully match both of our example pairs.

See Also

The documentation for the standard Text::Soundex and User::pwent modules; the Text::Metaphone and String::Approx modules from CPAN; your system's *passwd*(5) manpage; Volume 3, Chapter 6 of *The Art of Computer Programming*, by Donald E. Knuth (Addison-Wesley)

1.23 Program: fixstyle

Imagine you have a table with both old and new strings, such as the following:

Old words	New words
bonnet	hood
rubber	eraser
lorry	truck
trousers	pants

The program in Example 1-4 is a filter that changes all occurrences of each element in the first set to the corresponding element in the second set.

When called without filename arguments, the program is a simple filter. If filenames are supplied on the command line, an in-place edit writes the changes to the files, with the original versions saved in a file with a `".orig"` extension. See Recipe 7.16 for a description. A **-v** command-line option writes notification of each change to standard error.

The table of original strings and their replacements is stored below `__END__` in the main program, as described in Recipe 7.12. Each pair of strings is converted into carefully escaped substitutions and accumulated into the `$code` variable like the *popgrep2* program in Recipe 6.10.

A `-t` check to test for an interactive run check tells whether we're expecting to read from the keyboard if no arguments are supplied. That way if users forget to give an argument, they aren't wondering why the program appears to be hung.

Example 1-4. fixstyle

```
#!/usr/bin/perl -w
# fixstyle - switch first set of <DATA> strings to second set
#   usage: $0 [-v] [files ...]
use strict;
my $verbose = (@ARGV && $ARGV[0] eq '-v' && shift);
if (@ARGV) {
  $^I = ".orig";          # preserve old files
} else {
  warn "$0: Reading from stdin\n" if -t STDIN;
}
my $code = "while (<>) {\n";
# read in config, build up code to eval
while (<DATA>) {
  chomp;
  my ($in, $out) = split /\s*=>\s*/;
  next unless $in && $out;
  $code .= "s{\\Q$in\\E}{$out}g";
  $code .= "&& printf STDERR qq($in => $out at \$ARGV line \$.\\n)"
                                                      if $verbose;
  $code .= ";\n";
```

Example 1-4. fixstyle (continued)

```
}
$code .= "print;\n}\n";
eval "{ $code } 1" || die;
__END__
analysed        => analyzed
built-in        => builtin
chastized       => chastised
commandline     => command-line
de-allocate     => deallocate
dropin          => drop-in
hardcode        => hard-code
meta-data       => metadata
multicharacter  => multi-character
multiway        => multi-way
non-empty       => nonempty
non-profit      => nonprofit
non-trappable   => nontrappable
pre-define      => predefine
preextend       => pre-extend
re-compiling    => recompiling
reenter         => re-enter
turnkey         => turn-key
```

One caution: this program is fast, but it doesn't scale if you need to make hundreds of changes. The larger the DATA section, the longer it takes. A few dozen changes won't slow it down, and in fact, the version given in Example 1-4 is faster for that case. But if you run the program on hundreds of changes, it will bog down.

Example 1-5 is a version that's slower for few changes but faster when there are many changes.

Example 1-5. fixstyle2

```
#!/usr/bin/perl -w
# fixstyle2 - like fixstyle but faster for many many changes
use strict;
my $verbose = (@ARGV && $ARGV[0] eq '-v' && shift);
my %change = ();
while (<DATA>) {
  chomp;
  my ($in, $out) = split /\s*=>\s*/;
  next unless $in && $out;
  $change{$in} = $out;
}
if (@ARGV) {
  $^I = ".orig";
} else {
  warn "$0: Reading from stdin\n" if -t STDIN;
}
while (<>) {
  my $i = 0;
  s/^(\s+)// && print $1;         # emit leading whitespace
```

Example 1-5. fixstyle2 (continued)

```
  for (split /(\s+)/, $_, -1) {   # preserve trailing whitespace
      print( ($i++ & 1) ? $_ : ($change{$_} || $_));
  }
}
__END__
analysed        => analyzed
built-in        => builtin
chastized       => chastised
commandline     => command-line
de-allocate     => deallocate
dropin          => drop-in
hardcode        => hard-code
meta-data       => metadata
multicharacter  => multi-character
multiway        => multi-way
non-empty       => nonempty
non-profit      => nonprofit
non-trappable   => nontrappable
pre-define      => predefine
preextend       => pre-extend
re-compiling    => recompiling
reenter         => re-enter
turnkey         => turn-key
```

This version breaks each line into chunks of whitespace and words, which isn't a fast operation. It then uses those words to look up their replacements in a hash, which is much faster than a substitution. So the first part is slower, the second faster. The difference in speed depends on the number of matches.

If you don't care about keeping the whitespace separating each word constant, the second version can run as fast as the first, even for a few changes. If you know a lot about your input, collapse whitespace into single blanks by plugging in this loop:

```
# very fast, but whitespace collapse
while (<>) {
  for (split) {
      print $change{$_} || $_, " ";
  }
  print "\n";
}
```

That leaves an extra blank at the end of each line. If that's a problem, you could use the technique from Recipe 16.5 to install an output filter. Place the following code in front of the `while` loop that's collapsing whitespace:

```
my $pid = open(STDOUT, "|-");
die "cannot fork: $!" unless defined $pid;
unless ($pid) {             # child
      while (<STDIN>) {
      s/ $//;
      print;
  }
```

```
        exit;
    }
```

1.24 Program: psgrep

Many programs, including *ps*, *netstat*, *lsof*, *ls -l*, *find -ls*, and *tcpdump*, can produce more output than can be conveniently summarized. Logfiles also often grow too long to be easily viewed. You could send these through a filter like *grep* to pick out only certain lines, but regular expressions and complex logic don't mix well; just look at the hoops we jump through in Recipe 6.18.

What we'd really like is to make full queries on the program output or logfile. For example, to ask *ps* something like, "Show me all processes that exceed 10K in size but which aren't running as the superuser" or "Which commands are running on pseudo-ttys?"

The *psgrep* program does this—and infinitely more—because the specified selection criteria are not mere regular expressions; they're full Perl code. Each criterion is applied in turn to every line of output. Only lines matching all arguments are output. The following is a list of things to find and how to find them.

Lines containing "sh" at the end of a word:

```
% psgrep '/sh\b/'
```

Processes whose command names end in "sh":

```
% psgrep 'command =~ /sh$/'
```

Processes running with a user ID below 10:

```
% psgrep 'uid < 10'
```

Login shells with active ttys:

```
% psgrep 'command =~ /^-/' 'tty ne "?"'
```

Processes running on pseudo-ttys:

```
% psgrep 'tty =~ /^[p-t]/'
```

Non-superuser processes running detached:

```
% psgrep 'uid && tty eq "?"'
```

Huge processes that aren't owned by the superuser:

```
% psgrep 'size > 10 * 2**10' 'uid != 0'
```

The last call to *psgrep* produced the following output when run on our system. As one might expect, only *netscape* and its spawn qualified.

```
  FLAGS   UID   PID  PPID PRI  NI   SIZE   RSS WCHAN     STA TTY TIME COMMAND
      0   101  9751     1   0   0  14932  9652 do_select S   p1  0:25 netscape
 100000   101  9752  9751   0   0  10636   812 do_select S   p1  0:00 (dns helper)
```

Example 1-6 shows the *psgrep* program.

Example 1-6. psgrep

```
#!/usr/bin/perl -w
# psgrep - print selected lines of ps output by
#          compiling user queries into code
use strict;
# each field from the PS header
my @fieldnames = qw(FLAGS UID PID PPID PRI NICE SIZE
                  RSS WCHAN STAT TTY TIME COMMAND);
# determine the unpack format needed (hard-coded for Linux ps)
my $fmt = cut2fmt(8, 14, 20, 26, 30, 34, 41, 47, 59, 63, 67, 72);
my %fields;                        # where the data will store
die << Thanatos unless @ARGV;
usage: $0 criterion ...
  Each criterion is a Perl expression involving:
   @fieldnames
  All criteria must be met for a line to be printed.
Thanatos
# Create function aliases for uid, size, UID, SIZE, etc.
# Empty parens on
closure args needed for void prototyping.
for my $name (@fieldnames) {
  no strict 'refs';
  *$name = *{lc $name} = sub ( ) { $fields{$name} };
}
my $code = "sub is_desirable { " . join(" and ", @ARGV) . " } ";
unless (eval $code.1) {
  die "Error in code: $@\n\t$code\n";
}
open(PS, "ps wwaxl |")             || die "cannot fork: $!";
print scalar <PS>;                 # emit header line
while (<PS>) {
  @fields{@fieldnames} = trim(unpack($fmt, $_));
  print if is_desirable( );        # line matches their criteria
}
close(PS)                          || die "ps failed!";
# convert cut positions to unpack format
sub cut2fmt {
  my(@positions) = @_;
  my $template  = '';
  my $lastpos   = 1;
  for my $place (@positions) {
      $template .= "A" . ($place - $lastpos) . " ";
      $lastpos   = $place;
  }
  $template .= "A*";
  return $template;
}
sub trim {
  my @strings = @_;
  for (@strings) {
      s/^\s+//;
      s/\s+$//;
  }
  return wantarray ? @strings : $strings[0];
```

Example 1-6. psgrep (continued)

```
  }
  # the following was used to determine column cut points.
  # sample input data follows
  #123456789012345678901234567890123456789012345678901234567890123456789012345
  #         1         2         3         4         5         6         7
  # Positioning:
  #       8    14    20    26  30  34     41    47          59  63  67   72
  #       |     |     |     |   |   |      |     |           |   |   |    |
  __END__
 FLAGS   UID   PID  PPID PRI  NI   SIZE   RSS WCHAN       STA TTY TIME COMMAND
   100     0     1     0   0   0    760   432 do_select   S     ? 0:02 init
   140     0   187     1   0   0    784   452 do_select   S     ? 0:02 syslogd
100100   101   428     1   0   0   1436   944 do_exit     S     1 0:00 /bin/login
100140    99 30217   402   0   0   1552  1008 posix_lock_ S     ? 0:00 httpd
     0   101   593   428   0   0   1780  1260 copy_thread S     1 0:00 -tcsh
100000   101 30639  9562  17   0    924   496             R    p1 0:00 ps axl
     0   101 25145  9563   0   0   2964  2360 idetape_rea S    p2 0:06 trn
100100     0 10116  9564   0   0   1412   928 setup_frame T    p3 0:00 ssh -C www
100100     0 26560 26554   0   0   1076   572 setup_frame T    p2 0:00 less
100000   101 19058  9562   0   0   1396   900 setup_frame T    p1 0:02 nvi /tmp/a
```

The *psgrep* program integrates many techniques presented throughout this book. Stripping strings of leading and trailing whitespace is found in Recipe 1.19. Converting cut marks into an `unpack` format to extract fixed fields is in Recipe 1.1. Matching strings with regular expressions is the entire topic of Chapter 6.

The multiline string in the here document passed to `die` is discussed in Recipes 1.15 and 1.16. The assignment to `@fields{@fieldnames}` sets many values at once in the hash named `%fields`. Hash slices are discussed in Recipes 4.8 and 5.11.

The sample program input contained beneath `__END__` is described in Recipe 7.12. During development, we used canned input from the `DATA` filehandle for testing purposes. Once the program worked properly, we changed it to read from a piped-in *ps* command but left a remnant of the original filter input to aid in future porting and maintenance. Launching other programs over a pipe is covered in Chapter 16, including Recipes 16.10 and 16.13.

The real power and expressiveness in *psgrep* derive from Perl's use of string arguments not as mere strings but directly as Perl code. This is similar to the technique in Recipe 9.9, except that in *psgrep*, the user's arguments are wrapped with a routine called `is_desirable`. That way, the cost of compiling strings into Perl code happens only once, before the program whose output we'll process is even begun. For example, asking for UIDs under 10 creates this string to `eval`:

```
eval "sub is_desirable { uid < 10 } " . 1;
```

The mysterious `".1"` at the end is so that if the user code compiles, the whole `eval` returns true. That way we don't even have to check `$@` for compilation errors as we do in Recipe 10.12.

Specifying arbitrary Perl code in a filter to select records is a breathtakingly powerful approach, but it's not entirely original. Perl owes much to the *awk* programming language, which is often used for such filtering. One problem with *awk* is that it can't easily treat input as fixed-size fields instead of fields separated by something. Another is that the fields are not mnemonically named: *awk* uses `$1`, `$2`, etc. Plus, Perl can do much that *awk* cannot.

The user criteria don't even have to be simple expressions. For example, this call initializes a variable `$id` to user *nobody*'s number to use later in its expression:

```
% psgrep 'no strict "vars";
        BEGIN { $id = getpwnam("nobody") }
        uid == $id '
```

How can we use unquoted words without even a dollar sign, like `uid`, `command`, and `size`, to represent those respective fields in each input record? We directly manipulate the symbol table by assigning closures to indirect typeglobs, which creates functions with those names. The function names are created using both uppercase and lowercase names, allowing both `"UID < 10"` and `"uid > 10"`. Closures are described in Recipe 11.4, and assigning them to typeglobs to create function aliases is shown in Recipe 10.14.

One twist here not seen in those recipes is empty parentheses on the closure. These allowed us to use the function in an expression anywhere we'd use a single term, like a string or a numeric constant. It creates a void prototype so the field-accessing function named `uid` accepts no arguments, just like the built-in function `time`. If these functions weren't prototyped void, expressions like `"uid < 10"` or `"size/2 > rss"` would confuse the parser because it would see the unterminated start of a wildcard glob and of a pattern match, respectively. Prototypes are discussed in Recipe 10.11.

The version of *psgrep* demonstrated here expects the output from Red Hat Linux's *ps*. To port to other systems, look at which columns the headers begin at. This approach isn't relevant only to *ps* or only to Unix systems; it's a generic technique for filtering input records using Perl expressions, easily adapted to other record layouts. The input format could be in columns, space separated, comma separated, or the result of a pattern match with capturing parentheses.

The program could even be modified to handle a user-defined database with a small change to the selection functions. If you had an array of records as described in Recipe 11.9, you could let users specify arbitrary selection criteria, such as:

```
sub id()         { $_->{ID}   }
sub title()      { $_->{TITLE} }
sub executive()  { title =~ /(?:vice-)?president/i }

# user search criteria go in the grep clause
@slowburners = grep { id<10 && !executive } @employees;
```

For reasons of security and performance, this kind of power is seldom found in database engines like those described in Chapter 14. SQL doesn't support this, but given Perl and small bit of ingenuity, it's easy to roll it up on your own.