Chapter 14. Strings and Sorting

As we mentioned near the beginning of this book, Perl is designed to be good at solving programming problems that are about 90% working with text and 10% everything else. So it's no surprise that Perl has strong text-processing abilities, even without all that you've done with regular expressions. Sometimes the regular expression engine is too fancy and you need a simpler way of working with a string, as you'll see in this chapter.

Finding a Substring with index

Finding a substring depends on where you lost it. If you happen to have lost it within a bigger string, you're in luck because the `index` function can help you out. Here's how it looks:

```
my $where = index($big, $small);
```

Perl locates the first occurrence of the small string within the big string, returning an integer location of the first character. The character position returned is a zero-based value: if the substring is found at the very beginning of the string, `index` returns `0`; if it's one character later, the return value is `1`, and so on. If `index` can't find the substring at all, it returns `-1` to indicate that. In this example, `$where` gets `6` because that's the position where `wor` starts:

```
my $stuff = "Howdy world!";
my $where = index($stuff, "wor");
```

Another way you could think of the position number is the number of characters to skip over before getting to the substring. Since `$where` is `6`, you know that you have to skip over the first six characters of `$stuff` before you find `wor`.

The `index` function will always report the location of the *first found* occurrence of the substring. But you can tell it to start searching at a later point than the start of the string by using the optional third parameter, which tells `index` to start at that position:

```
my $stuff  = "Howdy world!";
my $where1 = index($stuff, "w");                # $where1 gets 2
my $where2 = index($stuff, "w", $where1 + 1);  # $where2 gets 6
my $where3 = index($stuff, "w", $where2 + 1);  # $where3 gets -1 (not found)
```

That third parameter is effectively giving a minimum value for the return value; if the substring isn't at that position or later, `index` returns `-1`. You probably wouldn't do this without a loop, though. In this example, we use an array to store the positions:

```
use v5.10;

my $stuff  = "Howdy world!";

my @where = ();
my $where = -1;
while( 1 ) {
  $where = index( $stuff, 'w', $where + 1 );
  last if $where == -1;
  push @where, $where;
}
say "Positions are @where";
```

We initialize `$where` to `-1` because we'll add `1` to it before we pass the starting position to `index`. This means that the first pass is not a special case.

Once in a while, you might prefer to have the last occurrence of the substring. You can get that with the `rindex` function, which starts scanning from the end of the string. In this example, you can find the last slash, which turns out to be at position `4` in the string, still counting from the left, just like `index`:

```
my $last_slash = rindex("/etc/passwd", "/");  # value is 4
```

The `rindex` function also has an optional third parameter, but in this case, it effectively gives the *maximum* permitted return value:

```
my $fred = "Yabba dabba doo!";

my $where1 = rindex($fred, "abba");  # $where1 gets 7
my $where2 = rindex($fred, "abba", $where1 - 1);  # $where2 gets 1
my $where3 = rindex($fred, "abba", $where2 - 1);  # $where3 gets -1
```

And here is its loop form. In this example, instead of starting at `-1`, we start at the position one beyond the last position. The length of the string is one greater than the zero-based last position:

```
use v5.10;

my $fred = "Yabba dabba doo!";

my @where = ();
my $where = length $fred;
while( 1 ) {
  $where = rindex($fred, "abba", $where - 1 );
  last if $where == -1;
  push @where, $where;
}
say "Positions are @where";
```

Manipulating a Substring with substr

The `substr` function works with only a part of a larger string. It looks like this:

```
my $part = substr($string, $initial_position, $length);
```

It takes three arguments: a string value, a zero-based initial position (such as the return value of `index`), and a length for the substring. The return value is the substring:

```
my $mineral = substr("Fred J. Flintstone", 8, 5);  # gets "Flint"
my $rock = substr "Fred J. Flintstone", 13, 1000;  # gets "stone"
```

The third argument to `substr` is the length of the substring you want. It's always going to be the length no matter how badly we want it to be the end position.

As you may have noticed in the previous example, if the requested length (`1000` characters, in this case) would go past the end of the string, Perl won't complain, but you get a shorter string than you might have expected. But if you want to be sure to go to the end of the string, however long or short it may be, just omit that third parameter (the length), like this:

```
my $pebble = substr "Fred J. Flintstone", 13;  # gets "stone"
```

The initial position of the substring in the larger string can be negative, counting from the end of the string (that is, position `-1` is the last character). In this example, position `-3` is three characters from the end of the string, which is the location of the letter `i`:

```
my $out = substr("some very long string", -3, 2);  # $out gets "in"
```

As you might expect, `index` and `substr` work well together. In this example, you can extract a substring that starts at the location of the letter `l`:

```
my $long = "some very very long string";
my $right = substr($long, index($long, "l") );
```

Now here's something really cool—you can change the selected portion of the string if the string is a variable:

```
my $string = "Hello, world!";
substr($string, 0, 5) = "Goodbye";  # "Goodbye, world!"
```

As you can see, the assigned (sub)string doesn't have to be the same length as the substring it's replacing. The string's length is adjusted to fit.

If you give it a length of 0, you can insert text without removing anything:

```
substr($string, 9, 0) = "cruel ";    # "Goodbye, cruel world!";
```

Or if that wasn't cool enough to impress you, you could use the binding operator (=~) to restrict an operation to work with just part of a string. This example replaces `fred` with `barney` wherever possible within just the last 20 characters of a string:

```
substr($string, -20) =~ s/fred/barney/g;
```

Much of the work that you do with `substr` and `index` you could also do with regular expressions. Use those where they're appropriate. But `substr` and `index` can often be faster, since they don't have the overhead of the regular expression engine: they're never case-insensitive, they have no metacharacters to worry about, and they don't set any of the capture variables.

Besides assigning to the `substr` function (which looks a little weird at first glance, perhaps), you can also use `substr` in a slightly more traditional manner with the four-argument version, in which the fourth argument is the replacement substring:

```
my $previous_value = substr($string, 0, 5, "Goodbye");
```

The previous value comes back as the return value, although as always, you can use this function in a void context to simply discard it.

Formatting Data with sprintf

The `sprintf` function takes the same arguments as `printf` (except for the optional filehandle, of course), but it returns the requested string instead of printing it. This is handy if you want to store a formatted string in a variable for later use, or if you want more control over the result than `printf` alone would provide:

```
my $date_tag = sprintf
  "%4d/%02d/%02d %2d:%02d:%02d",
  $yr, $mo, $da, $h, $m, $s;
```

In that example, `$date_tag` gets something like `"2038/01/19 3:00:08"`. The format string (the first argument to `sprintf`) used a leading zero on the format width number, which we didn't mention when we talked about `printf` formats in Chapter 5. The leading zero on the format number means to use leading zeros as needed to make the number as wide as requested. Without a leading zero in the formats, the resulting date-and-time string would have unwanted leading spaces instead of zeros, looking like `"2038/ 1/19 3: 0: 8"`.

Using sprintf with "Money Numbers"

One popular use for `sprintf` is when you want to format a number with a certain number of places after the decimal point, such as when you want to show an amount of money as `2.50` and not `2.5`—and certainly not as `2.49997`! That's easy to accomplish with the `"%.2f"` format:

```
my $money = sprintf "%.2f", 2.49997;
```

The full implications of rounding are numerous and subtle, but in most cases you should keep numbers in memory with all of the available accuracy, rounding off only for output.

If you have a "money number" that may be large enough to need commas to show its size, you might find it handy to use a subroutine like this one:

```
sub big_money {
  my $number = sprintf "%.2f", shift @_;
```

```
    # Add one comma each time through the do-nothing loop
    1 while $number =~ s/^(-?\d+)(\d\d\d)/$1,$2/;
    # Put the dollar sign in the right place
    $number =~ s/^(-?)/$1\$/;
    $number;
}
```

This subroutine uses some techniques you haven't seen yet, but they logically follow from what we've shown you. The first line of the subroutine formats the first (and only) parameter to have exactly two digits after the decimal point. That is, if the parameter were the number `12345678.9`, now your `$number` is the string `"12345678.90"`.

The next line of code uses a `while` modifier. As we mentioned when we covered that modifier in Chapter 10, that can always be rewritten as a traditional `while` loop:

```
while ($number =~ s/^(-?\d+)(\d\d\d)/$1,$2/) {
  1;
}
```

NOTE

In this example, we hardcoded the comma as the thousands separator. The `Number::Format` and `CLDR::Number` modules are more interesting to people who really care about these things.

What does that say to do? It says that as long as the substitution returns a true value (signifying success), the loop body should run. But the loop body does nothing! That's OK with Perl, but it tells us that the purpose of that statement is to do the conditional expression (the substitution) rather than the useless loop body. The value `1` is traditionally used as this kind of a placeholder, although any other value would be equally useful. This works just as well as the loop from before:

```
'keep looping' while $number =~ s/^(-?\d+)(\d\d\d)/$1,$2/;
```

So, now you know that the substitution is the real purpose of the loop. But what is the substitution doing? Remember that `$number` is some string like `"12345678.90"` at this point. The pattern will match the first part of the string, but it can't get past the decimal point. (Do you see why it can't?) Memory `$1` will get `"12345"`, and `$2` will get `"678"`, so the substitution will make `$number` into `"12345,678.90"` (remember, it couldn't match the decimal point, so the last part of the string is left untouched).

Do you see what the dash is doing near the start of that pattern? (Hint: the dash is allowed at only one place in the string.) We'll tell you at the end of this section, in case you haven't figured it out.

You're not done with that substitution statement yet. Since the substitution succeeded, the do-nothing loop goes back to try again. This time, the pattern can't match anything from the comma onward, so `$number` becomes `"12,345,678.90"`. The substitution thus adds a comma to the number each time through the loop.

Speaking of the loop, it's still not done. Since the previous substitution was a success, you're back around the loop to try again. But this time, the pattern can't match at all, since it has to match at least four digits at the start of the string, so now that is the end of the loop.

Why couldn't you have simply used the `/g` modifier to do a "global" search-and-replace, to save the trouble and confusion of the `1 while`? You couldn't use that because you're working backward from the decimal point rather than forward from the start of the string. You can't put the commas in a number like this simply with the `s///g` substitution alone. So, did you figure out the dash? It allows a possible minus sign at the start of the string. The next line of code makes the same allowance, putting the dollar sign in the right place so that `$number` is something like `"$12,345,678.90"`, or perhaps `"-$12,345,678.90"` if it's negative. Note that the dollar sign isn't necessarily the first character in the string, or that line would be a lot simpler. Finally, the last line of code returns your nicely formatted "money number," which you can print in the annual report.

Advanced Sorting

In Chapter 3, we showed that you could sort a list in ascending order by using the built-in `sort` operator. What if you want a numeric sort? Or a case-insensitive sort? Or maybe you want to sort items according to information stored in a hash. Well, Perl lets you sort a list in whatever order you'd need; you'll see all of those examples by the end of the chapter.

You'll tell Perl what order you want by making a *sort-definition subroutine*, or *sort subroutine* for short. Now, when you first hear the term "sort subroutine," if you've been through any computer science courses, visions of bubble sort and shell sort and quick sort race through your head, and you say, "No, never again!" Don't worry; it's not that bad. In fact, it's pretty simple. Perl already knows how to sort a list of items; it merely doesn't know which order you want. So the sort-definition subroutine simply tells it the order.

Why is this necessary? Well, if you think about it, sorting is putting a bunch of things in order by comparing them all. Since you can't compare them all at once, you need to compare two at a time, eventually using what you find out about each pair's order to put the whole kit and caboodle in line. Perl already understands all of those steps *except* for the part about how you'd like to compare the items, so that's all you have to write.

This means that the sort subroutine doesn't need to sort many items after all. It merely has to be able to compare two items. If it can put two items in the proper order, Perl will be able to tell (by repeatedly consulting the sort subroutine) what order you want for your data.

The sort subroutine is defined like an ordinary subroutine (well, almost). This routine will be called repeatedly, each time checking on a pair of elements from the list you're sorting.

Now, if you were writing a subroutine that's expecting to get two parameters that need sorting, you might write something like this to start:

```
sub any_sort_sub {    # It doesn't really work this way
  my($a, $b) = @_;    # Get and name the two parameters
  # start comparing $a and $b here
  ...
}
```

But you're going to call that sort subroutine again and again, often hundreds or thousands of times. Declaring the variables `$a` and `$b` and assigning them values at the top of the subroutine will take just a little time, but multiply that by the thousands of times you will call the routine and you can see that it contributes significantly to the overall execution speed.

You don't do it like that. (In fact, if you did it that way, it wouldn't work.) Instead, it is as if Perl has done this for you, before your subroutine's code has even started. You'll really write a sort subroutine without that first line; both `$a` and `$b` have been assigned for you. When the sort subroutine starts running, `$a` and `$b` are two elements from the original list.

The subroutine returns a coded value describing how the elements compare (like C's `qsort(3)` does, but it's Perl's own internal sort implementation). If `$a` should appear before `$b` in the final list, the sort subroutine returns `-1` to say so. If `$b` should appear before `$a`, it returns `1`.

If the order of `$a` and `$b` doesn't matter, the subroutine returns `0`. Why would it not matter? Perhaps you're doing a case-insensitive sort and the two strings are `fred` and `Fred`. Or perhaps you're doing a numeric sort and the two numbers are equal.

You could now write a numeric sort subroutine like this:

```
sub by_number {
  # a sort subroutine, expect $a and $b
  if ($a < $b) { -1 } elsif ($a > $b) { 1 } else { 0 }
}
```

To use the sort subroutine, just put its name (without an ampersand) between the keyword `sort` and the list you're sorting. This example puts a

numerically sorted list of numbers into `@result`:

```
my @result = sort by_number @some_numbers;
```

You can call this subroutine `by_number` to describe how it sorts. But more importantly, you can read the line of code that uses it with sort as saying "sort by number," as you would in English. Many people start their sort-subroutine names with `by_` to describe how they sort. Or you could have called this one `numerically` for a similar reason, but that's more typing and more chance to mess up something.

Notice that you don't have to do anything in the sort subroutine to declare `$a` and `$b` and set their values—and if you did, the subroutine wouldn't work right. We just let Perl set up `$a` and `$b` for us, so all you need to write is the comparison.

In fact, you can make it even simpler (and more efficient). Since this kind of three-way comparison is frequent, Perl has a convenient shortcut to use to write it. In this case, you use the spaceship operator (`<=>`). This operator compares two numbers and returns `-1`, `0`, or `1` as needed to sort them numerically. So you could write that sort subroutine better, like this:

```
sub by_number { $a <=> $b }
```

Since the spaceship compares numbers, you may have guessed that there's a corresponding three-way string-comparison operator: `cmp`. These two are easy to remember and keep straight. The spaceship has a family resemblance to the numeric comparison operators like `>=`, but it's three characters long instead of two because it has three possible return values instead of two. And `cmp` has a family resemblance to the string comparison operators like `ge`, but it's three characters long instead of two because it *also* has three possible return values instead of two. Of course, `cmp` by itself provides the same order as the default sort. You'd never need to write this subroutine, which yields merely the default sort order:

```
sub by_code_point { $a cmp $b }

my @strings = sort by_code_point @any_strings;
```

But you can use `cmp` to build a more complex sort order, like a case-insensitive sort:

```
sub case_insensitive { "\F$a" cmp "\F$b" }
```

In this case, you're comparing the string from `$a` (case-folded) against the string from `$b` (case-folded), giving a case-insensitive sort order.

But remember that Unicode has the concept of canonical and compatible equivalence, which we cover in Appendix C. To sort equivalent forms next to each other, you need to sort the decomposed form. If you are dealing with Unicode strings, this is probably what you want most of the time:

```
use Unicode::Normalize;

sub equivalents { NFKD($a) cmp NFKD($b) }
```

Note that you're not modifying the elements themselves in any of these; you're merely using their values. That's actually important: for efficiency reasons, `$a` and `$b` aren't copies of the data items. They're actually new, temporary aliases for elements of the original list, so if you change them, you mangle the original data. Don't do that—it's neither supported nor recommended.

When your sort subroutine is as simple as the ones you see here (and most of the time, it is), you can make the code even simpler yet, by replacing the name of the sort routine with the entire sort routine "inline," like so:

```
my @numbers = sort { $a <=> $b } @some_numbers;
```

In fact, in modern Perl, you'll hardly ever see a separate sort subroutine; you'll frequently find sort routines written inline as we've done here.

Suppose you want to sort in descending numeric order. That's easy enough to do with the help of `reverse`:

```
my @descending = reverse sort { $a <=> $b } @some_numbers;
```

But here's a neat trick. The comparison operators (`<=>` and `cmp`) are very nearsighted; that is, they can't see which operand is `$a` and which is `$b`, but only which *value* is on the left and which is on the right. So if `$a` and `$b` were to swap places, the comparison operator would get the results backward every time. That means that this is another way to get a reversed numeric sort:

```
my @descending = sort { $b <=> $a } @some_numbers;
```

You can (with a little practice) read this at a glance. It's a descending-order comparison (because `$b` comes before `$a`, which is descending order), and it's a numeric comparison (because it uses the spaceship instead of `cmp`). So it is sorting numbers in reverse order. (In modern Perl versions, it doesn't matter much which one of those you do, because `reverse` is recognized as a modifier to `sort`, and special shortcuts are taken to avoid sorting it one way just to have to turn it around the other way.)

Sorting a Hash by Value

Once you've been sorting lists happily for a while, you'll run into a situation where you want to sort a hash by value. For example, three of our characters went out bowling last night, and you have their bowling scores in the following hash. You want to be able to print out the list in the proper order, with the game winner at the top, so you have to sort the hash by score:

```
my %score = ("barney" => 195, "fred" => 205, "dino" => 30);
my @winners = sort by_score keys %score;
```

Of course, you aren't really going to be able to sort the hash by score; that's just a verbal shortcut. You can't sort a hash! But when you used `sort` with hashes before now, you sorted the keys of the hash (in code point order). Now, you're still going to sort the keys of the hash, but the order is now defined by their corresponding values from the hash. In this case, the result should be a list of our three characters' names, in order according to their bowling scores.

Writing this sort subroutine is fairly easy. What you want is to use a numeric comparison on the scores rather than the names. That is, instead of comparing `$a` and `$b` (the players' names), you want to compare `$score{$a}` and `$score{$b}` (their scores). If you think of it that way, it almost writes itself, as in:

```
sub by_score { $score{$b} <=> $score{$a} }
```

Step through this to see how it works. Imagine that the first time it's called, Perl has set `$a` to `barney` and `$b` to `fred`. So the comparison is `$score{"fred"} <=> $score{"barney"}`, which (as you can see by consulting the hash) is `205 <=> 195`. Remember, now, the spaceship is nearsighted, so when it sees `205` before `195`, it says, in effect: "No, that's not the right numeric order; `$b` should come before `$a`." So it tells Perl that `fred` should come before `barney`.

Maybe the next time the routine is called, `$a` is `barney` again but `$b` is now `dino`. The nearsighted numeric comparison sees `30 <=> 195` this time, so it reports that they're in the right order; `$a` does indeed sort in front of `$b`. That is, `barney` comes before `dino`. At this point, Perl has enough information to put the list in order: `fred` is the winner, then `barney` in second place, then `dino`.

Why did the comparison use the `$score{$b}` before the `$score{$a}`, instead of the other way around? That's because you want bowling scores

arranged in *descending* order, from the highest score of the winner down. So you can (again, after a little practice) read this one at sight as well: `$score{$b} <=> $score{$a}` means to sort according to the scores, in reversed numeric order.

Sorting by Multiple Keys

We forgot to mention that there was a fourth player bowling last night with the other three, so the hash really looked like this:

```
my %score = (
  "barney" => 195, "fred" => 205,
  "dino" => 30, "bamm-bamm" => 195,
);
```

Now, as you can see, `bamm-bamm` has the same score as `barney`. So which one will be first in the sorted list of players? There's no telling, because the comparison operator (seeing the same score on both sides) will have to return zero when checking those two.

Maybe that doesn't matter, but you generally prefer to have a well-defined sort. If several players have the same score, you want them to be together in the list, of course. But within that group, the names should be in code point order. How can you write the sort subroutine to say that? Again, this turns out to be pretty easy:

```
my @winners = sort by_score_and_name keys %score;

sub by_score_and_name {
  $score{$b} <=> $score{$a}  # by descending numeric score
    or
  $a cmp $b          # code point order by name
  }
```

How does this work? Well, if the spaceship sees two different scores, that's the comparison you want to use. It returns `-1` or `1`, a true value, so the low-precedence short-circuit `or` will mean the rest of the expres-

sion will be skipped, and the comparison you want is returned. (Remember, the short-circuit `or` returns the last expression evaluated.) But if the spaceship sees two identical scores, it returns `0`, a false value, and thus the `cmp` operator gets its turn at bat, returning an appropriate ordering value considering the keys as strings. That is, if the scores are the same, the string-order comparison breaks the tie.

You know that when you use the `by_score_and_name` sort subroutine like this, it will never return `0`, because no two hash keys are equal. So you know that the sort order is always well defined; that is, you know that the result today will be the same as the result with the same data tomorrow.

There's no reason that your sort subroutine has to be limited to two levels of sorting, of course. Here the Bedrock Library program puts a list of patron ID numbers in order according to a five-level sort. This example sorts according to the amount of each patron's outstanding fines (as calculated by a subroutine `&fines`, not shown here), the number of items they currently have checked out (from `%items`), their name (in order by family name, then by personal name, both from hashes), and finally by the patron's ID number, in case everything else is the same:

```
@patron_IDs = sort {
  &fines($b) <=> &fines($a) or
  $items{$b} <=> $items{$a} or
  $family_name{$a} cmp $family_name{$b} or
  $personal_name{$a} cmp $personal_name{$b} or
  $a <=> $b
} @patron_IDs;
```

Exercises

See "Answers to Chapter 14 Exercises" for answers to these exercises:

1. [10] Write a program to read in a list of numbers and sort them numerically, printing out the resulting list in a right-justified column. Try it out on this sample data:

```
17 1000 04 1.50 3.14159 -10 1.5 4 2001 90210 666
```

2. [15] Make a program that will print the following hash's data sorted in case-insensitive alphabetical order by last name. When the last names are the same, sort those by first name (again, without regard for case). That is, the first name in the output should be Fred's, while the last one should be Betty's. All of the people with the same family name should be grouped together. Don't alter the data. The names should be printed with the same capitalization as shown here:

```
my %last_name = qw{
  fred flintstone Wilma Flintstone Barney Rubble
  betty rubble Bamm-Bamm Rubble PEBBLES FLINTSTONE
};
```

3. [15] Make a program that looks through a given string for every occurrence of a given substring, printing out the positions where the substring is found. For example, given the input string `"This is a test."` and the substring `"is"`, it should report positions `2` and `5`. If the substring were `"a"`, it should report `8`. What does it report if the substring is `"t"`?