Appendix A. Exercise Answers

This appendix contains the answers to the exercises that appear throughout the book.

Answers to Chapter 1 Exercises

1. This exercise is easy since we already gave you the program. Your job was to make it work:

   ```
   print "Hello, world!\n";
   ```

   If you have v5.10 or later, you can try `say`:

   ```
   use v5.10;
   say "Hello, world!";
   ```

   If you want to try it from the command line without creating a file, you can specify your program on the command line with the `-e` switch:

   ```
   $ perl -e 'print "Hello, World\n"'
   ```

   There's another switch, `-l`, that automatically adds the newline for you:

   ```
   $ perl -le 'print "Hello, World"'
   ```

   The quoting in Windows in *command.exe* (or *cmd.exe*) needs the double quotes on the outside, so you switch them:

   ```
   C:\> perl -le "print 'Hello, World'"
   ```

You can save yourself some headaches with quoting by using the generalized quotes inside the shell quoting:

```
C:\> perl -le "print q(Hello, World)"
```

With v5.10 and later, you can use the `-E` switch to enable new features. That allows you to use `say`:

```
$ perl -E 'say q(Hello, World)'
```

We didn't expect you to try this on the command line, because we didn't tell you about that yet. Still, it's another way to do it. See perlrun for a complete listing of command-line switches and features.

2. The *perldoc* command should come with your *perl*, so you should be able to run it directly. If you can't find *perldoc*, you may have to install another package on your system. Ubuntu, for instance, puts it in the `perl-doc` package.
3. This program is easy too, as long as you got the previous exercise to work:

```
@lines = `perldoc -u -f atan2`;
foreach (@lines) {
  s/\w<(.+?)>/\U$1/g;
  print;
}
```

Answers to Chapter 2 Exercises

1. Here's one way to do it:

```
#!/usr/bin/perl
use warnings;
$pi = 3.141592654;
$circ = 2 * $pi * 12.5;
print "The circumference of a circle of radius 12.5 is $circ.\n";
```

As you see, we started this program with a typical `#!` line; your path to Perl may vary. We also turned on warnings.

The first real line of code sets the value of `$pi` to our value of π. There are several reasons a good programmer will prefer to use a constant value like this: it takes time to type `3.141592654` into your program if you ever need it more than once. It may be a mathematical bug if you accidentally used `3.141592654` in one place and `3.14159` in another. There's only one line to check on to make sure you didn't accidentally type `3.141952654` and send your space probe to the wrong planet.

Modern Perls allow you to use fancier characters as variable names. We could have used the π character as the name if we told Perl that the source code included Unicode characters (see Appendix C):

```
#!/usr/bin/perl
use utf8;
use warnings;
$π = 3.141592654;
$circ = 2 * $π * 12.5;
print "The circumference of a circle of radius 12.5 is $circ.\n";
```

Next we calculate the circumference, storing it in `$circ`, and we print it out in a nice message. The message ends with a newline character, because every line of a good program's output should end with a newline. Without it, you might end up with output looking something like this, depending on your shell's prompt:

```
The circumference of a circle of radius 12.5
is 78.53981635.bash-2.01$
```

Since the circumference isn't really `78.53981635.bash-2.01$`, this should probably be construed as a bug. So, use `\n` at the end of each line of output.

2. Here's one way to do it:

```
#!/usr/bin/perl
use warnings;
$pi = 3.141592654;
```

```
print "What is the radius? ";
chomp($radius = <STDIN>);
$circ = 2 * $pi * $radius;
print "The circumference of a circle of radius $radius is $circ.\n";
```

This is just like the last one, except now we ask the user for the radius, and then we use `$radius` in every place where we previously used the hardcoded value `12.5`. If we had written the first program with more foresight, in fact, we would have a variable named `$radius` in that one as well. Note that if we hadn't used `chomp`, the mathematical formula would still have worked because a string like `"12.5\n"` is converted to the number `12.5` without any problem. But when we print out the message, it would look like this:

```
The circumference of a circle of radius 12.5 is 78.53981635.
```

Notice that the newline character is still in `$radius`, even though we've used that variable as a number. Since we had a space between `$radius` and the word `is` in the `print` statement, there's a space at the beginning of the second line of output. The moral of the story is: `chomp` your input unless you have a reason not to.

3. Here's one way to do it:

```
#!/usr/bin/perl
use warnings;
$pi = 3.141592654;
print "What is the radius? ";
chomp($radius = <STDIN>);
$circ = 2 * $pi * $radius;
if ($radius < 0) {
  $circ = 0;
}
print "The circumference of a circle of radius $radius is $circ.\n";
```

Here we added the check for a bogus radius. Even if the given radius was impossible, the returned circumference would at least be nonnegative. You could have changed the given radius to be zero, and then calculated the circumference too; there's more than one way to do it.

In fact, that’s the Perl motto: There Is More Than One Way To Do It. And that’s why each exercise answer starts with “Here’s one way to do it…”

4. Here’s one way to do it:

```
print "Enter first number: ";
chomp($one = <STDIN>);
print "Enter second number: ";
chomp($two = <STDIN>);
$result = $one * $two;
print "The result is $result.\n";
```

Notice that we’ve left off the `#!` line for this answer. In fact, from here on, we’ll assume that you know it’s there, so you don’t need to read it each time.

Perhaps those are poor choices for variable names. In a large program, a maintenance programmer might think that `$two` should have the value of `2`. In this short program, it probably doesn’t matter, but in a large one we could have called them something more descriptive, with names like `$first_response`.

In this program, it wouldn’t make any difference if we forgot to `chomp` the two variables `$one` and `$two`, since we never use them as strings once they’ve been set. But if next week our maintenance programmer edits the program to print a message like: `The result of multiplying $one by $two is $result.\n`, those pesky newlines will come back to haunt us. Once again, `chomp` unless you have a reason not to `chomp`—like in the next exercise.

5. Here’s one way to do it:

```
print "Enter a string: ";
$str = <STDIN>;
print "Enter a number of times: ";
chomp($num = <STDIN>);
$result = $str x $num;
print "The result is:\n$result";
```

This program is almost the same as the last one, in a sense. We’re “multiplying” a string by a number of times, so we’ve kept the structure of the previous exercise. In this case, though, we didn't want to `chomp` the first input item—the string—because the exercise asked for the strings to appear on separate lines. So, if the user entered `fred` and a newline for the string, and `3` for the number, we'd get a newline after each `fred`, just as we wanted.

In the `print` statement at the end, we put the newline before `$result` because we wanted to have the first `fred` printed on a line of its own. That is, we didn't want output like this, with only two of the three `fred`s aligned in a column:

```
The result is: fred
fred
fred
```

At the same time, we didn't need to put another newline at the end of the `print` output because `$result` should already end with a newline.

In most cases, Perl won't mind where you put spaces in your program; you can put in spaces or leave them out. But it's important not to accidentally spell the wrong thing! If the `x` runs up against the preceding variable name `$str`, Perl will see `$strx`, which won't work.

Answers to Chapter 3 Exercises

1. Here's one way to do it:

```
print "Enter some lines, then press Ctrl-D:\n";
@lines = <STDIN>;
@reverse_lines = reverse @lines;
print @reverse_lines;
```

…or, even more simply:

```
print "Enter some lines, then press Ctrl-D:\n";
print reverse <STDIN>;
```

Most Perl programmers would prefer the second one, as long as they don’t need to keep the list of lines around for later use.

2. Here’s one way to do it:

```
@names = qw/ fred betty barney dino wilma pebbles bamm-bamm /;
print "Enter some numbers from 1 to 7, one per line, then press Ctrl-D:\n";
chomp(@numbers = <STDIN>);
foreach (@numbers) {
  print "$names[ $_ - 1 ]\n";
}
```

   We have to subtract one from the index number so that the user can count from 1 to 7, even though the array is indexed from 0 to 6. Another way to accomplish this would be to have a dummy item in the `@names` array, like this:

```
@names = qw/ dummy_item fred betty barney dino wilma pebbles bamm-bamm /;
```

   Give yourself extra credit if you checked to make sure that the user’s choice of index was in fact in the range 1 to 7.

3. Here’s one way to do it if you want the output all on one line:

```
chomp(@lines = <STDIN>);
@sorted = sort @lines;
print "@sorted\n";
```

   …or, to get the output on separate lines:

```
print sort <STDIN>;
```

Answers to Chapter 4 Exercises

1. Here’s one way to do it:

```
sub total {
  my $sum;  # private variable
  foreach (@_) {
    $sum += $_;
  }
  $sum;
}
```

This subroutine uses `$sum` to keep a running total. At the start of the subroutine, `$sum` is `undef`, since it's a new variable. Then the `foreach` loop steps through the parameter list (from `@_`), using `$_` as the control variable. (Note: once again, there's no automatic connection between `@_`, the parameter array, and `$_`, the default variable for the `foreach` loop.)

The first time through the `foreach` loop, the first number (in `$_`) is added to `$sum`. Of course, `$sum` is `undef`, since nothing has been stored in there. But since we're using it as a number, which Perl sees because of the numeric operator `+=`, Perl acts as if it's already initialized to `0`. Perl thus adds the first parameter to `0` and puts the total back in `$sum`.

The next time through the loop, the next parameter is added to `$sum`, which is no longer `undef`. The sum is placed back in `$sum`, and on through the rest of the parameters. Finally, the last line returns `$sum` to the caller.

There's a potential bug in this subroutine, depending on how you think of things. Suppose this subroutine was called with an empty parameter list (as we considered with the rewritten subroutine `&max` in the chapter text). In that case, `$sum` would be `undef`, and that would be the return value. But in this subroutine, it would probably be "more correct" to return `0` as the sum of the empty list, rather than `undef`. (Of course, if you wish to distinguish the sum of an empty list from the sum of, say, `(3, -5, 2)`, returning `undef` would be the right thing to do.)

If you don't want a possibly undefined return value, though, it's easy to remedy. Simply initialize `$sum` to zero rather than using the default of `undef`:

```
my $sum = 0;
```

Now the subroutine will always return a number, even if the parameter list was empty.

2. Here's one way to do it:

```
# Remember to include &total from previous exercise!
print "The numbers from 1 to 1000 add up to ", total(1..1000), ".\n";
```

Note that we can't call the subroutine from inside the double-quoted string, so the subroutine call is another separate item being passed to `print`. The total should be `500500`, a nice round number. And it shouldn't take any noticeable time at all to run this program; passing a parameter list of 1,000 values is an everyday task for Perl.

3. Here's one way to do it:

```
sub average {
  if (@_ == 0) { return }
  my $count = @_;
  my $sum = total(@_);               # from earlier exercise
  $sum/$count;
}

sub above_average {
  my $average = average(@_);
  my @list;
  foreach my $element (@_) {
    if ($element > $average) {
      push @list, $element;
    }
  }
  @list;
}
```

In `average`, we return without giving an explicit return value if the parameter list is empty. That gives the caller `undef` as the way to report that there's no average for an empty list. If the list wasn't empty, using `&total` makes it simple to calculate the average. We didn't even need to use temporary variables for `$sum` and `$count`, but doing so makes the code easier to read.

The second sub, `above_average`, simply builds up and returns a list of the desired items. (Why is the control variable named `$element`, instead of using Perl's favorite default, `$_`?) Note that this second sub uses a different technique for dealing with an empty parameter list.

4. To remember the last person that `greet` spoke to, use a `state` variable. It starts out as `undef`, which is how we figure out `Fred` is the first person it greets. At the end of the subroutine, we store the current `$name` in `$last_name` so we remember what it is next time:

```
use v5.10;

greet( 'Fred' );
greet( 'Barney' );

sub greet {
    state $last_person;

    my $name = shift;

    print "Hi $name! ";

    if( defined $last_person ) {
        print "$last_person is also here!\n";
    }
    else {
        print "You are the first one here!\n";
    }

    $last_person = $name;
}
```

5. This answer is similar to that for the preceding exercise, but this time we store all the names we have seen. Instead of using a scalar variable, we declare `@names` as a state variable and push each name onto it:

```
use v5.10;

greet( 'Fred' );
greet( 'Barney' );
greet( 'Wilma' );
greet( 'Betty' );

sub greet {
    state @names;

    my $name = shift;

    print "Hi $name! ";

    if( @names ) {
        print "I've seen: @names\n";
    }
    else {
        print "You are the first one here!\n";
    }

    push @names, $name;
}
```

Answers to Chapter 5 Exercises

1. Here's one way to do it:

   ```
   print reverse <>;
   ```

   Well, that's pretty simple! But it works because `print` is looking for a list of strings to print, which it gets by calling `reverse` in a list context. And `reverse` is looking for a list of strings to reverse, which it gets by using the diamond operator in a list context. So, the diamond returns a list of all the lines from all the files of the user's choice. That list of lines is just what *cat* would print out. Now `reverse` reverses the list of lines, and `print` prints them out.
2. Here's one way to do it:

```
print "Enter some lines, then press Ctrl-D:\n";  # or Ctrl-Z
chomp(my @lines = <STDIN>);

print "1234567890" x 7, "12345\n";  # ruler line to column 75

foreach (@lines) {
  printf "%20s\n", $_;
}
```

Here we start by reading in and chomping all of the lines of text. Then we print the ruler line. Since that's a debugging aid, we'd generally comment out that line when the program is done. We could have typed `"1234567890"` again and again, or even used copy-and-paste to make a ruler line as long as we needed, but we chose to do it this way because it's kind of cool.

Now the `foreach` loop iterates over the list of lines, printing each one with the `%20s` conversion. If you chose to do so, you could have created a format to print the list all at once, without the loop:

```
my $format = "%20s\n" x @lines;
printf $format, @lines;
```

It's a common mistake to get 19-character columns. That happens when you say to yourself, "Hey, why do we `chomp` the input if we're only going to add the newlines back on later?" So you leave out the `chomp` and use a format of `"%20s"` (without a newline). And now, mysteriously, the output is off by one space. So, what went wrong?

The problem happens when Perl tries to count the spaces needed to make the right number of columns. If the user enters `hello` and a newline, Perl sees *six* characters, not five, since newline is a character. So it prints 14 spaces and a six-character string, sure that it gives the 20 characters you asked for in `"%20s"`. Oops.

Of course, Perl isn't looking at the contents of the string to determine the width; it merely checks the raw number of characters. A newline (or another special character, such as a tab or a null character) will throw things off.

3. Here's one way to do it:

```
print "What column width would you like? ";
chomp(my $width = <STDIN>);

print "Enter some lines, then press Ctrl-D:\n";  # or Ctrl-Z
chomp(my @lines = <STDIN>);

print "1234567890" x (($width+9)/10), "\n";      # ruler line as needed

foreach (@lines) {
  printf "%${width}s\n", $_;
}
```

Instead of interpolating the width into the format string, we could have used this:

```
foreach (@lines) {
  printf "%*s\n", $width, $_;
}
```

This is much like the previous one, but we ask for a column width first. We ask for that first because we can't ask for more input *after* the end-of-file indicator, at least on some systems. Of course, in the real world you'll generally have a better end-of-input indicator when getting input from the user, as we'll see in later exercise answers.
Another change from the previous exercise's answer is the ruler line. We used some math to cook up a ruler line that's at least as long as we need, as suggested as an extra-credit part of the exercise. Proving that our math is correct is an additional challenge. (Hint: consider possible widths of `50` and `51`, and remember that the right side operand to `x` is truncated, not rounded.)
To generate the format this time, we used the expression `"%${width}s\n"`, which interpolates `$width`. The curly braces are required to "insulate" the name from the following `s`; without the curly braces, we'd be interpolating `$widths`, the wrong variable. If you forgot how to use curly braces to do this, though, you could have written an expression like `'%' . $width . "s\n"` to get the same format string.

The value of `$width` brings up another case where `chomp` is vital. If you don't `chomp` the width, the resulting format string would resemble `"%30\ns\n"`. That's not useful.
People who have seen `printf` before may have thought of another solution. Because `printf` comes to us from C, which doesn't have string interpolation, we can use the same trick that C programmers use. If an asterisk (`*`) appears in place of a numeric field width in a conversion, a value from the list of parameters will be used:

```
printf "%*s\n", $width, $_;
```

Answers to Chapter 6 Exercises

1. Here's one way to do it:

```
my %last_name = qw{
  fred flintstone
  barney rubble
  wilma flintstone
};
print "Please enter a first name: ";
chomp(my $name = <STDIN>);
print "That's $name $last_name{$name}.\n";
```

In this one, we used a `qw//` list (with curly braces as the delimiter) to initialize the hash. That's fine for this simple data set, and it's easy to maintain because each data item is a simple given name and simple family name, with nothing tricky. But if your data might contain spaces—for example, if `robert de niro` or `mary kay place` were to visit Bedrock—this simple method wouldn't work so well.
You might have chosen to assign each key-value pair separately, something like this:

```
my %last_name;
$last_name{"fred"} = "flintstone";
```

```
$last_name{"barney"} = "rubble";
$last_name{"wilma"} = "flintstone";
```

Note that (if you chose to declare the hash with `my`, perhaps because `use strict` was in effect) you must declare the hash before assigning any elements. You can't use `my` on only part of a variable, like this:

```
my $last_name{"fred"} = "flintstone";  # Oops!
```

The `my` operator works only with *entire* variables, never with just one element of an array or hash. Speaking of lexical variables, you may have noticed that the lexical variable `$name` is being declared inside the `chomp` function call; it is fairly common to declare each `my` variable as you need it, like this.

This is another case where `chomp` is vital. If someone enters the five-character string `"fred\n"` and we fail to `chomp` it, we'll be looking for `"fred\n"` as an element of the hash—and it's not there. Of course, `chomp` alone won't make this bulletproof; if someone enters `"fred \n"` (with a trailing space), with what we've seen so far, we don't have a way to tell that they meant `fred`.

If you added a check for whether the given key `exists` in the hash so that you'll give the user an explanatory message when they misspell a name, give yourself extra points for that.

2. Here's one way to do it:

```
my(@words, %count, $word);     # (optionally) declare our variables
chomp(@words = <STDIN>);

foreach $word (@words) {
  $count{$word} += 1;          # or $count{$word} = $count{$word} + 1;
}

foreach $word (keys %count) {  # or sort keys %count
  print "$word was seen $count{$word} times.\n";
}
```

In this one, we declared all of the variables at the top. People who come to Perl from a background in languages like Pascal (where variables are always declared "at the top") may find that way more familiar than declaring variables as they are needed. Of course, we're declaring these because we're pretending that `use strict` may be in effect; by default, Perl won't require such declarations.

Next we use the line-input operator, `<STDIN>`, in a list context to read all of the input lines into `@words`, and then we `chomp` those all at once. So `@words` is our list of words from the input (if the words were all on separate lines, as they should have been, of course).

Now the first `foreach` loop goes through all the words. That loop contains the most important statement of the entire program, the statement that says to add one to `$count{$word}` and put the result back in `$count{$word}`. Although you could write it either the short way (with the `+=` operator) or the long way, the short way is just a little bit more efficient, since Perl has to look up `$word` in the hash just once.

For each word in the first `foreach` loop, we add one to `$count{$word}`. So, if the first word is `fred`, we add one to `$count{"fred"}`. Of course, since this is the first time we've seen `$count{"fred"}`, it's `undef`. But since we're treating it as a number (with the numeric `+=` operator, or with `+` if you wrote it the long way), Perl converts `undef to 0` for us automatically. The total is `1`, which is then stored back in `$count{"fred"}`.

The next time through that `foreach` loop, let's say the word is `barney`. So, we add one to `$count{"barney"}`, bumping it up from `undef to 1` as well.

Now let's say the next word is `fred` again. When we add one to `$count{"fred"}`, which is already `1`, we get `2`. This goes back in `$count{"fred"}`, meaning that we've now seen `fred` twice.

When we finish the first `foreach` loop, then, we've counted how many times each word has appeared. The hash has a key for each (unique) word from the input, and the corresponding value is the number of times that word appeared.

So now, the second `foreach` loop goes through the keys of the hash, which are the unique words from the input. In this loop, we'll see each *different* word once. For each one, it says something like "`fred was seen 3 times.`"

If you want the extra credit on this problem, you could put `sort` before `keys` to print out the keys in order. If there will be more than a dozen items in an output list, it's generally a good idea for them to be sorted so that a human being who is trying to debug the program will fairly quickly be able to find the item they want.

3. Here's one way to do it:

```
my $longest = 0;
foreach my $key ( keys %ENV ) {
    my $key_length = length( $key );
    $longest = $key_length if $key_length > $longest;
    }

foreach my $key ( sort keys %ENV ) {
    printf "%-${longest}s  %s\n", $key, $ENV{$key};
    }
```

In the first `foreach` loop, we go through all of the keys and use `length` to get their lengths. If the length we just measured is greater than the one we stored in `$longest`, we put the longer value in `$longest`.

Once we've gone through all of the keys, we use `printf` to print the keys and values in two columns. We use the same trick we used in Exercise 3 from Chapter 5 by interpolating `$longest` into the template string.

Answers to Chapter 7 Exercises

1. Here's one way to do it:

```
while (<>) {
  if (/fred/) {
    print;
  }
}
```

This is pretty simple. The more important part of this exercise is trying it out on the sample strings. It doesn't match `Fred`, showing that regular expressions are case-sensitive. (We'll see how to change that later.) It does match `frederick` and `Alfred`, since both of those strings contain the four-letter string `fred`. (Matching whole words only, so that `frederick` and `Alfred` wouldn't match, is another feature we'll see later.)

2. Here's one way to do it: change the pattern used in the first exercise's answer to `/[fF]red/`. You could also have tried `/(f|F)red/` or `/fred|Fred/`, but the character class is more efficient.
3. Here's one way to do it: change the pattern used in the first exercise's answer to `/\./`. The backslash is needed because the dot is a metacharacter, or you could use a character class: `/[.]/`.
4. Here's one way to do it: change the pattern used in the first exercise's answer to `/[A-Z][a-z]+/`.
5. Here's one way to do it: change the pattern used in the first exercise's answer to `/(\S)\1/`. The `\S` character class matches the nonwhitespace character, and the parentheses allow you to use the back reference `\1` to match the same character immediately following it.
6. Here's one way to do it:

```
while (<>) {
  if (/wilma/) {
    if (/fred/) {
      print;
    }
  }
}
```

This tests `/fred/` only after we find `/wilma/` matches, but `fred` could appear before or after `wilma` in the line; each test is independent of the other.

If you wanted to avoid the extra nested `if` test, you might have written something like this:

```
while (<>) {
  if (/wilma.*fred|fred.*wilma/) {
    print;
    }
  }
```

This works because you'll either have `wilma` before `fred` or `fred` before `wilma`. If we had written just `/wilma.*fred/`, that wouldn't have matched a line like `fred and wilma flintstone`, even though that line mentions both of them.

Folks who know about the logical-and operator, which we showed in Chapter 10, could do both tests `/fred/` and `/wilma/` in the same `if` conditional. That's more efficient, more scalable, and an all-around better way than the ones given here. But we haven't seen logical-and yet:

```
  while (<>) {
    if (/wilma/ && /fred/) {
      print;
    }
  }
```

The low-precedence short-circuit version works too:

```
  while (<>) {
    if (/wilma/ and /fred/) {
      print;
    }
  }
```

We made this an extra-credit exercise because many folks have a mental block here. We showed you an "or" operation (with the vertical bar, `|`), but we never showed you an "and" operation. That's because there isn't one in regular expressions. *Mastering Perl* revisits this example by using a regular expression lookahead, something even a bit too advanced for *Intermediate Perl*.

Answers to Chapter 8 Exercises

1. There's one easy way to do it, and we showed it back in the chapter body. But if your output isn't saying `before<match>after` as it should, you've chosen a hard way to do it.
2. Here's one way to do it:

   ```
   /a\b/
   ```

   (Of course, that's a pattern for use inside the pattern test program!) If your pattern mistakenly matches `barney`, you probably needed the word-boundary anchor.
3. Here's one way to do it:

   ```
   #!/usr/bin/perl
   while (<STDIN>) {
     chomp;
     if (/(\b\w*a\b)/) {
       print "Matched: |$`<$&>$'|\n";
       print "\$1 contains '$1'\n";       # The new output line
     } else {
       print "No match: |$_|\n";
     }
   }
   ```

   This is the same test program (with a new pattern), except that the one marked line has been added to print out `$1`.
   The pattern uses a pair of `\b` word-boundary anchors inside the parentheses, although the pattern works the same way when they are placed outside. That's because anchors correspond to a place in the string but not to any characters in the string: anchors have "zero width."
   Admittedly, the first `\b` anchor isn't really needed, due to details about greediness that we won't go into here. But it may help a tiny bit with efficiency, and it certainly helps with clarity—and in the end, that one wins out.
4. This exercise answer is the same as the previous exercise answer, but with a slightly different regular expression:

```
#!/usr/bin/perl

use v5.10;

while (<STDIN>) {
  chomp;
  if (/(?<word>\b\w*a\b)/) {
    print "Matched: |$`<$&>$'|\n";
    print "'word' contains '$+{word}'\n";       # The new output line
  } else {
    print "No match: |$_|\n";
  }
}
```

5. Here’s one way to do it:

```
m!
  (\b\w+a\b)       # $1: a word ending in a
  (.{0,5})         # $2: up to five characters following
!xs                # /x and /s modifiers
```

   (Don’t forget to add code to display `$2`, now that you have two memory variables. If you change the pattern to have just one again, you can simply comment out the extra line.) If your pattern doesn’t match just plain `wilma` anymore, perhaps you require zero or more characters instead of one or more. You may have omitted the `/s` modifier, since there shouldn’t be newlines in the data. (Of course, if there are newlines in the data, the `/s` modifier could make for different output.)

6. Here’s one way to do it:

```
while (<>) {
  chomp;
  if (/\s\z/) {
    print "$_#\n";
  }
}
```

   We used the pound sign (`#`) as the marker character.

Answers to Chapter 9 Exercises

1. Here’s one way to do it:

   ```
   /($what){3}/
   ```

   Once `$what` has been interpolated, this gives a pattern resembling `/(fred| barney){3}/`. Without the parentheses, the pattern would be something like `/fred|barney{3}/`, which is the same as `/fred|barneyyy/`. So the parentheses are required.
2. Here’s one way to do it:

   ```
   my $in = $ARGV[0];
   if (! defined $in) {
     die "Usage: $0 filename";
   }

   my $out = $in;
   $out =~ s/(\.\w+)?$/.out/;

   if (! open $in_fh, '<', $in ) {
     die "Can't open '$in': $!";
   }

   if (! open $out_fh, '>', $out ) {
     die "Can't write '$out': $!";
   }

   while (<$in_fh>) {
     s/Fred/Larry/gi;
     print $out_fh $_;
   }
   ```

   This program begins by naming its one and only command-line parameter, and complaining if it didn’t get it. Then it copies that to `$out` and does a substitution to change the file extension, if any, to `.out`. (It would be sufficient, though, to merely append `.out` to the filename.)

Once the filehandles `$in_fh` and `$out_fh` are opened, the real program can begin. If you didn't use both options `/g` and `/i`, take off half a point, since *every* `fred` and `Fred` should be changed.

3. Here's one way to do it:

   ```
   while (<$in_fh>) {
     chomp;
     s/Fred/\n/gi;          # Replace all FREDs
     s/Wilma/Fred/gi;       # Replace all WILMAs
     s/\n/Wilma/g;          # Replace the placeholder
     print $out_fh "$_\n";
   }
   ```

   This replaces the loop from the previous program, of course. To do this kind of a swap, we need to have some "placeholder" string that doesn't otherwise appear in the data. By using `chomp` (and adding the newline back for the output), we ensure that a newline (`\n`) can be the placeholder. (You could choose some other unlikely string as the placeholder. Another good choice would be the NUL character, `\0`.)

4. Here's one way to do it:

   ```
   $^I = ".bak";          # make backups
   while (<>) {
     if (/\A#!/) {         # is it the shebang line?
       $_ .= "## Copyright (C) 20XX by Yours Truly\n";
     }
     print;
   }
   ```

   Invoke this program with the filenames you want to update. For example, if you've been naming your exercises *ex01-1*, *ex01-2*, and so on, so that they all begin with `ex...`, you would use:

   ```
   ./fix_my_copyright ex*
   ```

5. To keep from adding the copyright twice, we have to make two passes over the files. First, we'll make a "set" with a hash where the keys are

the filenames and the values don't matter (although we'll use `1` for convenience):

```
my %do_these;
foreach (@ARGV) {
  $do_these{$_} = 1;
}
```

Next, we'll examine the files and remove from our to-do list any file that already contains the copyright. The current filename is in `$ARGV`, so we can use that as the hash key:

```
while (<>) {
  if (/\A## Copyright/) {
    delete $do_these{$ARGV};
  }
}
```

Finally, it's the same program as before, once we've reestablished a re-duced list of names in `@ARGV`:

```
@ARGV = sort keys %do_these;
$^I = ".bak";           # make backups
exit unless @ARGV; # no arguments reads from standard input!
while (<>) {
  if (/\A#!/) {         # is it the shebang line?
    $_ .= "## Copyright (c) 20XX by Yours Truly\n";
  }
  print;
}
```

Answers to Chapter 10 Exercises

1. Here's one way to do it:

```
my $secret = int(1 + rand 100);
# This next line may be uncommented during debugging
# print "Don't tell anyone, but the secret number is $secret.\n";

while (1) {
  print "Please enter a guess from 1 to 100: ";
  chomp(my $guess = <STDIN>);
  if ($guess =~ /quit|exit|\A\s*\z/i) {
    print "Sorry you gave up. The number was $secret.\n";
    last;
  } elsif ($guess < $secret) {
    print "Too small. Try again!\n";
  } elsif ($guess == $secret) {
    print "That was it!\n";
    last;
  } else {
    print "Too large. Try again!\n";
  }
}
```

The first line picks out our secret number from `1` to `100`. Here's how it works. First, `rand` is Perl's random number function, so `rand 100` gives us a random number in the range from `0` up to (but not including) `100`. That is, the largest possible value of that expression is something like `99.999`. Adding one gives a number from `1` to `100.999`, then the `int` function truncates that, giving a result from `1` to `100`, as we needed.

The commented-out line can be helpful during development and debugging, or if you like to cheat. The main body of this program is the infinite `while` loop. That will keep asking for guesses until we execute `last`.

It's important that we test the possible strings before the numbers. If we didn't, do you see what would happen when the user types `quit`? That would be interpreted as a number (probably giving a warning message, if warnings were turned on), and since the value as a number would be zero, the poor user would get the message that their guess was too small. We might never get to the string tests, in that case.

Another way to make the infinite loop here would be to use a naked block with `redo`. It's not more or less efficient; merely another way to write it. Generally, if you expect to mostly loop, it's good to write `while`, since that loops by default. If looping will be the exception, a naked block may be a better choice.

2. This program is a slight modification to the previous answer. We want to print the secret number while we are developing the program, so we `print` the secret number if the variable `$Debug` has a true value. The value of `$Debug` is either the value that we already set as an environment variable, or `1` by default. By using the `//` operator, we won't set it to `1` unless the `$ENV{DEBUG}` is undefined:

   ```
   use v5.10;

   my $Debug = $ENV{DEBUG} // 1;

   my $secret = int(1 + rand 100);

   print "Don't tell anyone, but the secret number is $secret.\n"
       if $Debug;
   ```

   To do this without features introduced in v5.10, we just have to do a little more work:

   ```
   my $Debug = defined $ENV{DEBUG} ? $ENV{DEBUG} : 1;
   ```

3. Here's one way to do it, which steals from the answer to Exercise 3 in Chapter 6.
   At the top of the program, we set some environment variables. The keys `ZERO` and `EMPTY` have false but defined values, and the key `UNDEFINED` has no value.
   Later, in the `printf` argument list, we use the `//` operator to select the string `(undefined value)` only when `$ENV{$key}` is not a defined value:

   ```
   use v5.10;

   $ENV{ZERO}      = 0;
   $ENV{EMPTY}     = '';
   $ENV{UNDEFINED} = undef;
   ```

```
my $longest = 0;
foreach my $key ( keys %ENV ) {
  my $key_length = length( $key );
  $longest = $key_length if $key_length > $longest;
}

foreach my $key ( sort keys %ENV ) {
  printf "%-${longest}s  %s\n", $key, $ENV{$key} // "(undefined value)";
}
```

By using `//` here, we don't disturb false values such as those in the keys `ZERO` and `EMPTY`.
To do this without Perl 5.10, we use the ternary operator instead:

```
printf "%-${longest}s  %s\n", $key,
    defined $ENV{$key} ? $ENV{$key} : "(undefined value)";
```

Answers to Chapter 11 Exercises

1. This answer uses a hash reference (which you'll have to read about in *Intermediate Perl*), but we gave you the part to get around that. You don't have to know how it all works as long as you know it does work. You can get the job done and learn the details later.
   Here's one way to do it:

```
use Module::CoreList;

my %modules = %{ $Module::CoreList::version{5.034} };

print join "\n", keys %modules;
```

And here's a bonus. With Perl's `postderef` feature, you could write this:

```
use v5.20;
use feature qw(postderef);
no warnings qw(experimental::postderef);
```

```
use Module::CoreList;

my %modules = $Module::CoreList::version{5.034}->%*;

print join "\n", keys %modules;
```

See the blog post "Use postfix dereferencing" for more information. Or wait until we release the third edition of *Intermediate Perl*, which we will update with this new feature. We'll start working on it right after we finish this book.

2. Once you install `Time::Moment` from CPAN, you just have to create two dates and subtract them from each other. Remember to get the date order correct:

```
use Time::Moment;

my $now = Time::Moment->now;

my $then = Time::Moment->new(
    year       => $ARGV[0],
    month      => $ARGV[1],
    );

my $years  = $then->delta_years( $now );
my $months = $then->delta_months( $now ) % 12;

printf "%d years and %d months\n", $years, $months;
```

Answers to Chapter 12 Exercises

1. Here's one way to do it:

```
foreach my $file (@ARGV) {
  my $attribs = &attributes($file);
  print "'$file' $attribs.\n";
}
```

```
sub attributes {
  # report the attributes of a given file
  my $file = shift @_;
  return "does not exist" unless -e $file;

  my @attrib;
  push @attrib, "readable" if -r $file;
  push @attrib, "writable" if -w $file;
  push @attrib, "executable" if -x $file;
  return "exists" unless @attrib;
  'is ' . join " and ", @attrib;  # return value
}
```

In this solution, once again it's convenient to use a subroutine. The main loop prints one line of attributes for each file, perhaps telling us that `'cereal-killer' is executable` or that `'sasquatch' does not exist`.
The subroutine tells us the attributes of the given filename. Of course, if the file doesn't even exist, there's no need for the other tests, so we test for that first. If there's no file, we'll return early.
If the file does exist, we'll build a list of attributes. (Give yourself extra-credit points if you used the special `_` filehandle instead of `$file` on these tests, to keep from calling the system separately for each new attribute.) It would be easy to add additional tests like the three we show here. But what happens if none of the attributes is true? Well, if we can't say anything else, at least we can say that the file exists, so we do. The `unless` clause uses the fact that `@attrib` will be true (in a Boolean context, which is a special case of a scalar context) if it's got any elements.
But if we've got some attributes, we'll join them with `" and "` and put `"is "` in front, to make a description like `is readable and writable`. This isn't perfect, however; if there are three attributes, it says the file `is readable and writable and executable`, which has too many `and`s, but we can get away with it. If you wanted to add more attributes to the ones this program checks for, you should probably fix it to say something like `is readable, writable, executable, and nonempty`. If that matters to you.

Note that if you somehow didn't put any filenames on the command line, this produces no output. This makes sense; if you ask for information on zero files, you should get zero lines of output. But let's compare that to what the next program does in a similar case, in the explanation that follows.

2. Here's one way to do it:

```
die "No file names supplied!\n" unless @ARGV;
my $oldest_name = shift @ARGV;
my $oldest_age = -M $oldest_name;

foreach (@ARGV) {
  my $age = -M;
  ($oldest_name, $oldest_age) = ($_, $age)
    if $age > $oldest_age;
}

printf "The oldest file was %s, and it was %.1f days old.\n",
  $oldest_name, $oldest_age;
```

   This one starts right out by complaining if it didn't get any filenames on the command line. That's because it's supposed to tell us the oldest filename—and there ain't one if there aren't any files to check.
   Once again, we're using the "high-water mark" algorithm. The first file is certainly the oldest one seen so far. We have to keep track of its age as well so that's in `$oldest_age`.
   For each of the remaining files, we'll determine the age with the `-M` file test, just as we did for the first one (except that here we'll use the default argument of `$_` for the file test). The last-modified time is generally what people mean by the "age" of a file, although you could make a case for using a different one. If the age is more than `$oldest_age`, we'll use a list assignment to update both the name and age. We didn't have to use a list assignment, but it's a convenient way to update several variables at once.
   We stored the age from `-M` in the temporary variable `$age`. What would have happened if we had simply used `-M` each time, rather than using a variable? Well, first, unless we used the special `_` filehandle, we would have been asking the operating system for the age

of the file each time, a potentially slow operation (not that you'd notice unless you have hundreds or thousands of files, and maybe not even then). More importantly, though, we should consider what would happen if someone updated a file while we were checking it. That is, first we see the age of some file, and it's the oldest one seen so far. But before we can get back to use `-M` a second time, someone modifies the file and resets the timestamp to the current time. Now the age that we save into `$oldest_age` is actually the *youngest* age possible. The result would be that we'd get the oldest file among the files tested from that point on, rather than the oldest overall; this would be a tough problem to debug!

Finally, at the end of the program, we use `printf` to print out the name and age, with the age rounded off to the nearest tenth of a day. Give yourself extra credit if you went to the trouble to convert the age to a number of days, hours, and minutes.

3. Here's one way to do it:

```
use v5.10;

say "Looking for my files that are readable and writable";

die "No files specified!\n" unless @ARGV;

foreach my $file ( @ARGV ) {
  say "$file is readable and writable" if -o -r -w $file;
}
```

To use stacked file test operators, we need to use Perl 5.10 or later, so we start with the `use` statement to ensure that we have the right version of Perl. We `die` if there are no elements in `@ARGV`, and go through them with `foreach` otherwise.

We have to use three file test operators: `-o` to check if we own the file, `-r` to check that it is readable, and `-w` to check if it is writable. Stacking them as `-o -r -w` creates a composite test that only passes if all three of them are true, which is exactly what we want.

If we wanted to do this with a version before Perl 5.10, it's just a little more code. The `say`s become `print`s with added newlines, and the stacked file tests become separate tests combined with the `&&` short-circuit operator:

```
print "Looking for my files that are readable and writable\n";

die "No files specified!\n" unless @ARGV;

foreach my $file ( @ARGV ) {
  print "$file is readable and writable\n"
    if( -w $file && -r _ && -o _ );
  }
```

Answers to Chapter 13 Exercises

1. Here’s one way to do it, with a `glob`:

```
print "Which directory? (Default is your home directory) ";
chomp(my $dir = <STDIN>);
if ($dir =~ /\A\s*\z/) {         # A blank line
  chdir or die "Can't chdir to your home directory: $!";
} else {
  chdir $dir or die "Can't chdir to '$dir': $!";
}

my @files = <*>;
foreach (@files) {
  print "$_\n";
}
```

First, we show a simple prompt, read the desired directory, and `chomp` it. (Without a `chomp`, we’d be trying to head for a directory that ends in a newline—legal in Unix, and therefore cannot be presumed to simply be extraneous by the `chdir` function.)
Then, if the directory name is nonempty, we’ll change to that directory, aborting on a failure. If empty, the home directory is selected instead.
Finally, a `glob` on “star” pulls up all the names in the (new) working directory, automatically sorted in alphabetical order, and they’re printed one at a time.

2. Here's one way to do it:

```
print "Which directory? (Default is your home directory) ";
chomp(my $dir = <STDIN>);
if ($dir =~ /\A\s*\z/) {         # A blank line
  chdir or die "Can't chdir to your home directory: $!";
} else {
  chdir $dir or die "Can't chdir to '$dir': $!";
}

my @files = <.* *>;       ## now includes .*
foreach (sort @files) {   ## now sorts
  print "$_\n";
}
```

   Two differences from previous one. First, the `glob` now includes "dot star," which matches all the names that *do* begin with a dot. And second, we now must sort the resulting list because some of the names that begin with a dot must be interleaved appropriately, either before or after the list of things, without a beginning dot.

3. Here's one way to do it:

```
print 'Which directory? (Default is your home directory) ';
chomp(my $dir = <STDIN>);
if ($dir =~ /\A\s*\z/) {         # A blank line
  chdir or die "Can't chdir to your home directory: $!";
} else {
  chdir $dir or die "Can't chdir to '$dir': $!";
}

opendir DOT, "." or die "Can't opendir dot: $!";
foreach (sort readdir DOT) {
  # next if /\A\./; ##   if we were skipping dot files
  print "$_\n";
}
```

   Again, same structure as the previous two programs, but now we've chosen to open a directory handle. Once we've changed the working directory, we want to open the current directory, and we've shown that as the `DOT` directory handle.

Why `DOT`? Well, if the user asks for an absolute directory name, like `/etc`, there's no problem opening it. But if the name is relative, like `fred`, let's see what would happen. First, we `chdir` to `fred`, and then we want to use `opendir` to open it. But that would open `fred` in the new directory, not `fred` in the original directory. The only name we can be sure will mean "the current directory" is "`.`", which always has that meaning (in Unix and similar systems, at least).

The `readdir` function pulls up all the names of the directory, which are then sorted and displayed. If we had done the first exercise this way, we would have skipped over the dot files—and that's handled by uncommenting the commented-out line in the `foreach` loop.

You may find yourself asking, "Why did we `chdir` first? You can use `readdir` and friends on any directory, not merely on the current directory." Primarily, we wanted to give the user the convenience of being able to get to their home directory with a single keystroke. But this could be the start of a general file-management utility program; maybe the next step would be to ask the user which of the files in this directory should be moved to offline tape storage, say.

4. Here's one way to do it:

```
unlink @ARGV;
```

   …or, if you want to warn the user of any problems:

```
foreach (@ARGV) {
  unlink $_ or warn "Can't unlink '$_': $!, continuing...\n";
}
```

   Here, each item from the command-invocation line is placed individually into `$_`, which is then used as the argument to `unlink`. If something goes wrong, the warning gives a clue about why.

5. Here's one way to do it:

```
use File::Basename;
use File::Spec;

my($source, $dest) = @ARGV;

if (-d $dest) {
  my $basename = basename $source;
  $dest = File::Spec->catfile($dest, $basename);
}

rename $source, $dest
  or die "Can't rename '$source' to '$dest': $!\n";
```

The workhorse in this program is the last statement, but the remainder of the program is necessary when we are renaming into a directory. First, after declaring the modules we're using, we name the command-line arguments sensibly. If `$dest` is a directory, we need to extract the basename from the `$source` name and append it to the directory (`$dest`). Finally, once `$dest` is patched up if needed, the `rename` does the deed.

6. Here's one way to do it:

```
use File::Basename;
use File::Spec;

my($source, $dest) = @ARGV;

if (-d $dest) {
  my $basename = basename $source;
  $dest = File::Spec->catfile($dest, $basename);
}

link $source, $dest
  or die "Can't link '$source' to '$dest': $!\n";
```

As the hint in the exercise description said, this program is much like the previous one. The difference is that we'll `link` rather than `rename` . If your system doesn't support hard links, you might have written this as the last statement:

```
    print "Would link '$source' to '$dest'.\n";
```

7. Here’s one way to do it:

   ```
   use File::Basename;
   use File::Spec;

   my $symlink = $ARGV[0] eq '-s';
   shift @ARGV if $symlink;

   my($source, $dest) = @ARGV;
   if (-d $dest) {
     my $basename = basename $source;
     $dest = File::Spec->catfile($dest, $basename);
   }

   if ($symlink) {
     symlink $source, $dest
       or die "Can't make soft link from '$source' to '$dest': $!\n";
   } else {
     link $source, $dest
       or die "Can't make hard link from '$source' to '$dest': $!\n";
   }
   ```

   The first few lines of code (after the two `use` declarations) look at the first command-line argument, and if it’s `-s` , we’re making a symbolic link, so we note that as a true value for `$symlink` . If we saw that `-s` , we then need to get rid of it (in the next line). The next few lines are cut-and-pasted from the previous exercise answers. Finally, based on the truth of `$symlink` , we’ll choose to create either a symbolic link or a hard link. We also updated the dying words to make it clear which kind of link we were attempting.
8. Here’s one way to do it:

   ```
   foreach ( glob( '.* *' ) ) {
     my $dest = readlink $_;
     print "$_ -> $dest\n" if defined $dest;
   }
   ```

Each item resulting from the `glob` ends up in `$_` one by one. If the item is a symbolic link, then `readlink` returns a defined value, and the location is displayed. If not, the condition fails and we skip over it.

Answers to Chapter 14 Exercises

1. Here's one way to do it:

```
my @numbers;
push @numbers, split while <>;
foreach (sort { $a <=> $b } @numbers) {
  printf "%20g\n", $_;
}
```

That second line of code is too confusing, isn't it? Well, we did that on purpose. Although we recommend that you write clear code, some people like writing code that's as hard to understand as possible, so we want you to be prepared for the worst. Someday you'll need to maintain confusing code like this.

Since that line uses the `while` modifier, it's the same as if it were written in a loop like this:

```
while (<>) {
  push @numbers, split;
}
```

That's better, but maybe it's still a little unclear. (Nevertheless, we don't have a quibble about writing it this way. This one is on the correct side of the "too hard to understand at a glance" line.) The `while` loop is reading the input one line at a time (from the user's choice of input sources, as shown by the diamond operator), and `split` is, by default, splitting that on whitespace to make a list of words—or in this case, a list of numbers. The input is just a stream of numbers separated by whitespace, after all. Either way you write it, then, that `while` loop will put all of the numbers from the input into `@numbers` .

The `foreach` loop takes the sorted list and prints each item on its own line, using the `%20g` numeric format to put them in a right-justified column. You could have used `%20s` instead. What difference would that make? Well, that's a string format, so it would have left the strings untouched in the output. Did you notice that our sample data included both `1.50` and `1.5`, and both `04` and `4`? If you printed those as strings, the extra zero characters will still be in the output; but `%20g` is a numeric format, so equal numbers will appear identically in the output. Either format could potentially be correct, depending on what you're trying to do.

2. Here's one way to do it:

```
# don't forget to incorporate the hash %last_name,
# either from the exercise text or the downloaded file

my @keys = sort {
  "\L$last_name{$a}" cmp "\L$last_name{$b}"  # by last name
    or
  "\L$a" cmp "\L$b"                          # by first name
} keys %last_name;

foreach (@keys) {
  print "$last_name{$_}, $_\n";              # Rubble,Bamm-Bamm
}
```

There's not much to say about this one; we put the keys in order as needed, then print them out. We chose to print them in last-name-comma-first-name order just for fun; the exercise description left that up to you.

3. Here's one way to do it:

```
print "Please enter a string: ";
chomp(my $string = <STDIN>);
print "Please enter a substring: ";
chomp(my $sub = <STDIN>);

my @places;

for (my $pos = -1; ; ) {             # tricky use of three-part for loop
```

```
    $pos = index($string, $sub, $pos + 1);  # find next position
    last if $pos == -1;
    push @places, $pos;
  }

  print "Locations of '$sub' in '$string' were: @places\n";
```

This one starts out simply enough, asking the user for the strings and declaring an array to hold the list of substring positions. But once again, as we see in the `for` loop, the code seems to have been "optimized for cleverness," which should be done only for fun, never in production code. But this actually shows a valid technique, which could be useful in some cases, so let's see how it works.

The `my` variable `$pos` is declared private to the scope of the `for` loop, and it starts with a value of `-1`. So as not to keep you in suspense about this variable, we'll tell you right now that it's going to hold a position of the substring in the larger string. The test and increment sections of the `for` loop are empty, so this is an infinite loop. (Of course, we'll eventually break out of it, in this case with `last`.)

The first statement of the loop body looks for the first occurrence of the substring at or after position `$pos + 1`. That means that on the first iteration, when `$pos` is still `-1`, the search will start at position `0`, the start of the string. The location of the substring is stored back in `$pos`. Now, if that was `-1`, we're done with the `for` loop, so `last` breaks out of the loop in that case. If it wasn't `-1`, then we save the position into `@places` and go around the loop again. This time, `$pos + 1` means that we'll start looking for the substring just after the previous place where we found it. And so we get the answers we wanted and the world is once again a happy place.

If you didn't want that tricky use of the `for` loop, you could accomplish the same result as shown here:

```
  {
    my $pos = -1;
    while (1) {
      ... # Same loop body as the for loop used earlier
    }
  }
```

The naked block on the outside restricts the scope of `$pos`. You don't have to do that, but it's often a good idea to declare each variable in the smallest possible scope. This means we have fewer variables "alive" at any given point in the program, making it less likely that we'll accidentally reuse the name `$pos` for some new purpose. For the same reason, if you don't declare a variable in a small scope, you should generally give it a longer name that's thereby less likely to be reused by accident. Maybe something like `$substring_position` would be appropriate in this case.

On the other hand, if you were *trying* to obfuscate your code (shame on you!), you could create a monster like this (shame on us!):

```
for (my $pos = -1; -1 !=
  ($pos = index
    +$string,
    +$sub,
    +$pos
    +1
  );
push @places, ((((+$pos))))) {
    'for ($pos != 1; # ;$pos++) {
      print "position $pos\n";#;';#' } pop @places;
}
```

That even trickier code works in place of the original tricky `for` loop. By now, you should know enough to be able to decipher that one on your own, or to obfuscate code in order to amaze your friends and confound your enemies. Be sure to use these powers only for good, never for evil.

Oh, and what did you get when you searched for `t` in `This is a test.`? It's at positions `10` and `13`. It's not at position `0`; since the capitalization doesn't match, the substring doesn't match.

Answers to Chapter 15 Exercises

1. Here's one way to do it:

```
chdir '/' or die "Can't chdir to root directory: $!";
exec 'ls', '-l' or die "Can't exec ls: $!";
```

The first line changes the current working directory to the root directory, as our particular hardcoded directory. The second line uses the multiple-argument `exec` function to send the result to standard output. We could have used the single-argument form just as well, but it doesn't hurt to do it this way.

2. Here's one way to do it:

```
open STDOUT, '>', 'ls.out' or die "Can't write to ls.out: $!";
open STDERR, '>', 'ls.err' or die "Can't write to ls.err: $!";
chdir '/' or die "Can't chdir to root directory: $!";
exec 'ls', '-l' or die "Can't exec ls: $!";
```

The first and second lines reopen `STDOUT` and `STDERR` to a file in the current directory (before we change directories). Then, after the directory change, the directory listing command executes, sending the data back to the files opened in the original directory.

Where would the message from the last `die` go? Well, it would go into *ls.err*, of course, since that's where `STDERR` is going at that point. The `die` from `chdir` would go there too. But where would the message go if we can't reopen `STDERR` on the second line? It goes to the old `STDERR`. When reopening the three standard filehandles (`STDIN`, `STDOUT`, and `STDERR`), the old filehandles are still open.

3. Here's one way to do it:

```
if (`date` =~ /\AS/) {
  print "go play!\n";
} else {
  print "get to work!\n";
}
```

Well, since both Saturday and Sunday start with an S, and the day of the week is the first part of the output of the *date* command, this is pretty simple. Just check the output of the *date* command to see if it starts with `S`. There are many harder ways to do this program, and we've seen most of them in our classes.

If we had to use this in a real-world program, though, we'd probably use the pattern `/\A(Sat|Sun)/`. It's a tiny bit less efficient, but that hardly matters; besides, it's so much easier for the maintenance programmer to understand.

4. To catch some signals, we set up signal handlers. Just with the techniques we show in this book, we have a bit of repetitive work to do. In each handler subroutine, we set up a `state` variable so we can count the number of times we call that subroutine. We use a `foreach` loop to then assign the right subroutine name to the appropriate key in `%SIG`. At the end, we create an infinite loop so the program runs indefinitely:

```
use v5.10;

sub my_hup_handler  { state $n; say 'Caught HUP: ',  ++$n }
sub my_usr1_handler { state $n; say 'Caught USR1: ', ++$n }
sub my_usr2_handler { state $n; say 'Caught USR2: ', ++$n }
sub my_int_handler  { say 'Caught INT. Exiting.'; exit }

say "I am $$";

foreach my $signal ( qw(int hup usr1 usr2) ) {
    $SIG{ uc $signal } = "my_${signal}_handler";
    }

while(1) { sleep 1 };
```

We need another terminal session to run a program to send the signals:

```
$ kill -HUP 61203
$ perl -e 'kill HUP => 61203'
$ perl -e 'kill USR2 => 61203'
```

The output shows the running count of signals as we catch them:

```
$ perl signal_catcher
I am 61203
Caught HUP: 1
Caught HUP: 2
Caught USR2: 1
Caught HUP: 3
Caught USR2: 2
Caught INT. Exiting.
```

Answers to Chapter 16 Exercises

1. Here's one way to do it:

```
my $filename = 'path/to/sample_text';
open my $fh, '<', $filename
  or die "Can't open '$filename': $!";
chomp(my @strings = <$fh>);
while (1) {
  print 'Please enter a pattern: ';
  chomp(my $pattern = <STDIN>);
  last if $pattern =~ /\A\s*\Z/;
  my @matches = eval {
    grep /$pattern/, @strings;
  };
  if ($@) {
    print "Error: $@";
  } else {
    my $count = @matches;
    print "There were $count matching strings:\n",
      map "$_\n", @matches;
  }
  print "\n";
}
```

This one uses an `eval` block to trap any failure that might occur when using the regular expression. Inside that block, a `grep` pulls the matching strings from the list of strings.
Once the `eval` is finished, we can report either the error message or the matching strings. Note that we "unchomped" the strings for output

by using `map` to add a newline to each string.

2. This program is simple. There are many ways that we can get a list of files, but since we only care about the ones in the current working directory we can just use a `glob`. We use `foreach` to put each filename in the default variable `$_` since we know that `stat` uses that variable by default. We surround the entire `stat` before we perform the slice:

   ```
   foreach ( glob( '*' ) ) {
     my( $atime, $mtime ) = (stat)[8,9];
     printf "%-20s %10d %10d\n", $_, $atime, $mtime;
     }
   ```

   We know to use the indices 8 and 9 because we look at the documentation for `stat`. The documentation writers have been quite kind to us by showing us a table that maps the index of the list item to what it does, so we don't have to count over ourselves.
   If we don't want to use `$_`, we can use our own control variable:

   ```
   foreach my $file ( glob( '*' ) ) {
     my( $atime, $mtime ) = (stat $file)[8,9];
     printf "%-20s %10d %10d\n", $file, $atime, $mtime;
     }
   ```

3. This solution builds on the previous one. The trick now is to use `localtime` to turn the epoch times into date strings in the form YYYY-MM-DD. Before we integrate that into the full program, let's look at how we would do that, assuming that the time is in `$_` (which is the `map` control variable).
   We get the indices for the slice from the `localtime` documentation:

   ```
   my( $year, $month, $day ) = (localtime)[5,4,3];
   ```

   We note that `localtime` returns the year minus 1900 and the month minus 1 (at least minus 1 how we humans count), so we have to adjust that:

```
$year += 1900; $month += 1;
```

Finally, we can put it all together to get the format we want, padding the month and day with zeros if necessary:

```
sprintf '%4d-%02d-%02d', $year, $month, $day;
```

To apply this to a list of times, we use a `map`. Note that `localtime` is one of the operators that doesn't use `$_` by default, so you have to supply it as an argument explicitly:

```
my @times = map {
  my( $year, $month, $day ) = (localtime($_))[5,4,3];
  $year += 1900; $month += 1;
  sprintf '%4d-%02d-%02d', $year, $month, $day;
  } @epoch_times;
```

This, then, is what we have to substitute in our `stat` line in the previous program, finally ending up with:

```
foreach my $file ( glob( '*' ) ) {
  my( $atime, $mtime ) = map {
    my( $year, $month, $day ) = (localtime($_))[5,4,3];
    $year  += 1900; $month += 1;
    sprintf '%4d-%02d-%02d', $year, $month, $day;
    } (stat $file)[8,9];

  printf "%-20s %10s %10s\n", $file, $atime, $mtime;
  }
```

Most of the point of this exercise was to use the particular techniques we covered in Chapter 16. There's another way to do this, though, and it's much easier. The `POSIX` module, which comes with Perl, has a `strftime` subroutine that takes a `sprintf`-style format string and the time components in the same order that `localtime` returns them. That makes the `map` much simpler:

```
use POSIX qw(strftime);

foreach my $file ( glob( '*' ) ) {
  my( $atime, $mtime ) = map {
    strftime( '%Y-%m-%d', localtime($_) );
    } (stat $file)[8,9];

  printf "%-20s %10s %10s\n", $file, $atime, $mtime;
  }
```