Chapter 5. Input and Output

You've already seen how to do some input/output (I/O) in order to make some of the earlier exercises possible. But now you'll learn more about those operations by covering the 80% of the I/O you'll need for most programs. If you're already familiar with the workings of standard input, output, and error streams, you're ahead of the game. If not, we'll get you caught up by the end of this chapter. For now, just think of "standard input" as being "the keyboard," and "standard output" as being "the display screen."

Input from Standard Input

Reading from the standard input stream is easy. You've been doing it already with the `<STDIN>` operator. Evaluating this operator in a scalar context gives you the next line of input:

```
$line = <STDIN>;                # read the next line
chomp($line);                   # and chomp it

chomp($line = <STDIN>);         # same thing, more idiomatically
```

What we're calling the line-input operator here, `<STDIN>`, is actually a line-input operator (represented by the angle brackets) around a *filehandle*. You'll learn about filehandles later in this chapter.

Since the line-input operator will return `undef` when you reach end-of-file, this is handy for dropping out of loops:

```
while (defined($line = <STDIN>)) {
  print "I saw $line";
}
```

There's a lot going on in that first line: you're reading the input into a variable, checking that it's defined, and if it is (meaning that we haven't reached the end of the input), you're running the body of the `while` loop. So, inside the body of the loop, you'll see each line, one after another, in `$line`. This is something you'll want to do fairly often, so naturally Perl has a shortcut for it. The shortcut looks like this:

```
while (<STDIN>) {
  print "I saw $_";
}
```

Now, to make this shortcut, Larry chose some useless syntax. That is, this is *literally* saying, "Read a line of input, and see if it's true. (Normally it is.) And if it is true, enter the `while` loop, but *throw away that line of input!*" Larry knew that it was a useless thing to do; nobody should ever need to do that in a real Perl program. So Larry took this useless syntax and made it useful.

What this is *actually* saying is that Perl should do the same thing as you saw in our earlier loop: it tells Perl to read the input into a variable, and (as long as the result was defined, so you haven't reached end-of-file) then enter the `while` loop. However, instead of storing the input in `$line`, Perl uses its favorite default variable, `$_`, just as if you had written this:

```
while (defined($_ = <STDIN>)) {
  print "I saw $_";
}
```

Now, before you go any further, we must be very clear about something: this shortcut works *only* if you write it just like that. If you put a line-input operator anywhere else (in particular, as a statement all on its own), it won't read a line into `$_` by default. It works *only* if there's nothing but the line-input operator in the conditional of a `while` loop. If you put anything else into the conditional expression, this shortcut won't apply.

There's otherwise no other connection between the line-input operator (`<STDIN>`) and Perl's favorite default variable (`$_`). In this case, though,

it just happens that Perl is storing the input in that variable.

On the other hand, evaluating the line-input operator in a list context gives you all of the (remaining) lines of input as a list—each element of the list is one line:

```
foreach (<STDIN>) {
  print "I saw $_";
}
```

Once again, there's no connection between the line-input operator and Perl's favorite default variable. In this case, though, the default control variable for `foreach` is `$_`. So in this loop, you see each line of input in `$_`, one after the other.

That may sound familiar, and for good reason: that's the same thing the `while` loop would do. Isn't it?

The difference is under the hood. In the `while` loop, Perl reads a single line of input, puts it into a variable, and runs the body of the loop. Then it goes back to find another line of input. But in the `foreach` loop, you're using the line-input operator in a list context (since `foreach` needs a list to iterate through); you read all of the input before the loop can start running. That difference will become apparent when the input is coming from your 400 MB web server logfile! It's generally best to use code like the `while` loop's shortcut, which will process input a line at a time, whenever possible.

Input from the Diamond Operator

Another way to read input is with the diamond operator: `<>`. This is useful for making programs that work like standard Unix utilities, with respect to the invocation arguments (which we'll see in a moment). If you want to make a Perl program that can be used like the utilities *cat*, *sed*, *awk*, *sort*, *grep*, *lpr*, and many others, the diamond operator will be your

friend. If you want to make anything else, the diamond operator probably won't help.

NOTE

Randal went over to Larry's house one day to show off the new training materials he'd been writing and complained that there was no spoken name for "that thing." Larry didn't have a name for it either. Heidi (eight years old at the time) quickly chimed in, "That's a diamond, Daddy." So the name stuck. Thanks, Heidi!

The *invocation arguments* to a program are normally a number of "words" on the command line after the name of the program. In this case, they give the names of the files your program will process in sequence:

```
$ ./my_program fred barney betty
```

That command means to run the command *my_program* (which will be found in the current directory), and that it should process file *fred*, followed by file *barney*, followed by file *betty*.

If you give no invocation arguments, the program should process the standard input stream. Or, as a special case, if you give just a hyphen as one of the arguments, that means standard input as well. So, if the invocation arguments had been *fred - betty*, that would have meant that the program should process file *fred*, followed by the standard input stream, followed by file *betty*.

The benefit of making your programs work like this is that you may choose where the program gets its input at runtime; for example, you won't have to rewrite the program to use it in a pipeline (which we'll show more later). Larry put this feature into Perl because he wanted to make it easy for you to write your own programs that work like standard Unix utilities—even on non-Unix machines. Actually, he did it so he could make his *own* programs work like standard Unix utilities; since some vendors' utilities don't work just like others', Larry could make his own utilities, deploy them on a number of machines, and know that they'd all have

the same behavior. Of course, this meant porting Perl to every machine he could find.

The diamond operator is actually a special kind of line-input operator. But instead of getting the input from the keyboard, it comes from the user's choice of input:

```
while (defined($line = <>)) {
  chomp($line);
  print "It was $line that I saw!\n";
}
```

So, if you run this program with the invocation arguments `fred`, `barney`, and `betty`, it will say something like: "It was [a line from file fred] that I saw!", "It was [another line from file fred] that I saw!", on and on until it reaches the end of file *fred*. Then it will automatically go on to file *barney*, printing out one line after another, and then on through file *betty*. Note that there's no break when you go from one file to another; when you use the diamond, it's as if the input files have been merged into one big file. The diamond will return `undef` (and we'll drop out of the `while` loop) only at the end of all of the input.

NOTE

If it matters to you, or even if it doesn't, the current file's name is kept in Perl's special variable `$ARGV`. This name may be `"-"` instead of a real filename if the input is coming from the standard input stream, though.

In fact, since this is just a special kind of line-input operator, you may use the same shortcut you saw earlier to read the input into `$_` by default:

```
while (<>) {
  chomp;
  print "It was $_ that I saw!\n";
}
```

This works like the loop from before but with less typing. And you may have noticed that you use the default for `chomp`; without an argument, `chomp` works on `$_`. Every little bit of saved typing helps!

Since you generally use the diamond operator to process all of the input, it's typically a mistake to use it in more than one place in your program. If you find yourself putting two diamonds into the same program, especially using the second diamond inside the `while` loop that is reading from the first one, it's almost certainly not going to do what you would like. In our experience, when beginners put a second diamond into a program, they meant to use `$_` instead. Remember, the diamond operator *reads* the input, but the input itself is (generally, by default) found in `$_`.

If the diamond operator can't open one of the files and read from it, it'll print an allegedly helpful diagnostic message, such as:

```
can't open wilma: No such file or directory
```

The diamond operator will then go on to the next file automatically, much like what you'd expect from *cat* or another standard utility.

The Double Diamond

There's a problem with the diamond operator that has a fix in v5.22. Suppose that the filename from the command line has a special character in it, such as `|`. This might cause *perl* to perform a "pipe open" (see Chapter 15), which runs an external program and reads that program's output as if it were a file. The "double diamond" operator, `<<>>`, prevents this. It's the same as the diamond operator you just saw but without the magic that will run external programs:

```
use v5.22;

while (<<>>) {
  chomp;
  print "It was $_ that I saw!\n";
}
```

If you have v5.22 or later, you should use this version instead. We might have liked someone to fix the good ol' single diamond operator, but that might break something that someone has relied on for years. Instead, the Perl developers maintained backward compatibility.

For the rest of this book, we'll say "diamond operator," leaving it up to you to choose which one you want to use. We'll use the old single diamond to drag along the people on older versions.

The Invocation Arguments

Technically, the diamond operator isn't looking literally at the invocation arguments—it works from the `@ARGV` array. This array is a special array that is preset by the Perl interpreter as the list of the invocation arguments. In other words, this is just like any other array (except for its funny, all-caps name), but when your program starts, `@ARGV` is already stuffed full of the list of invocation arguments.

You can use `@ARGV` just like any other array; you could `shift` items off of it, perhaps, or use `foreach` to iterate over it. You could even check to see if any arguments start with a hyphen so that you could process them as invocation options (like Perl does with its own `-w` option).

NOTE

If you need more than just one or two such options, you should almost certainly use a module to process them in a standard way. See the documentation for the `Getopt::Long` and `Getopt::Std` modules, which are part of the standard distribution.

The diamond operator looks in `@ARGV` to determine what filenames it should use. If it finds an empty list, it uses the standard input stream; otherwise, it uses the list of files that it finds. This means that after your program starts and before you start using the diamond, you've got a chance to tinker with `@ARGV`. For example, you can process three specific files, regardless of what the user chose on the command line:

```
@ARGV = qw# larry moe curly #;  # force these three files to be read
while (<>) {
  chomp;
  print "It was $_ that I saw in some stooge-like file!\n";
}
```

Output to Standard Output

The `print` operator takes a list of values and sends each item (as a string, of course) to standard output in turn, one after another. It doesn't add any extra characters before, after, or in between the items; if you want spaces between items and a newline at the end, you have to say so:

```
$name = "Larry Wall";
print "Hello there, $name, did you know that 3+4 is ", 3+4, "?\n";
```

Of course, that means that there's a difference between printing an array and interpolating an array:

```
print @array;     # print a list of items
print "@array";   # print a string (containing an interpolated array)
```

That first `print` statement will print a list of items, one after another, with no spaces in between. The second one will print exactly one item, which is the string you get by interpolating `@array` into the empty string —that is, it prints the contents of `@array`, separated by spaces. So, if `@array` holds `qw/ fred barney betty /`, the first one prints `fredbarneybetty`, while the second prints `fred barney betty` separated by spaces. But before you decide to always use the second form, imagine that `@array` is a list of unchomped lines of input. That is, imagine that each of its strings has a trailing newline character. Now, the first `print` statement prints `fred`, `barney`, and `betty` on three separate lines. But the second one prints this:

```
fred
 barney
 betty
```

Do you see where the spaces come from? Perl is interpolating an array, so it puts spaces between the elements (actually, whatever is in the variable `$"`). So we get the first element of the array (`fred` and a newline character), then a space, then the next element of the array (`barney` and a newline character), then a space, then the last element of the array (`betty` and a newline character). The result is that the lines seem to have become indented, except for the first one.

Every week or two, we encounter a question like “Perl indents everything after the first line.” Without even reading the message, we can immediately see that the program used double quotes around an array containing unchomped strings. “Did you perhaps put an array of unchomped strings inside double quotes?” we ask, and the answer is always “yes.”

Generally, if your strings contain newlines, you simply want to print them, after all:

```
print @array;
```

But if they don’t contain newlines, you generally want to add one at the end:

```
print "@array\n";
```

So, if you use the quote marks, you’re (generally) adding the `\n` at the end of the string anyway; this should help you remember which is which.

It’s normal for your program’s output to be *buffered*. That is, instead of sending out every little bit of output at once, your program saves the output until there’s enough to bother with.

If, for example, you want to save the output to disk, it's (relatively) slow and inefficient to spin the disk every time you add one or two characters to the file. Generally, then, the output will go into a buffer that is *flushed* (that is, actually written to disk, or wherever) only when the buffer gets full, or when the output is otherwise finished (such as at the end of run-time). Usually, that's what you want.

But if you (or a program) may be waiting impatiently for the output, you may wish to take that performance hit and flush the output buffer each time you `print`. See `$!` in the perlvar documentation for more information on controlling buffering in that case.

Since `print` is looking for a list of strings to print, it evaluates its arguments in list context. Since the diamond operator (as a special kind of line-input operator) returns a list of lines in a list context, these can work well together:

```
print <>;          # implementation of /bin/cat

print sort <>;     # implementation of /bin/sort
```

Well, to be fair, the standard Unix commands *cat* and *sort* do have some additional functionality that these replacements lack. But you can't beat them for the price! You can now reimplement all of your standard Unix utilities in Perl and painlessly port them to any machine that has Perl, whether that machine is running Unix or not. And you can be sure that the programs on every different type of machine will nevertheless have the same behavior.

NOTE

The Perl Power Tools project, whose goal is to implement all of the classic Unix utilities in Perl, completed nearly all the utilities. Perl Power Tools has been helpful because it has made these standard utilities available on many non-Unix machines.

What might not be obvious is that `print` has optional parentheses, which can sometimes cause confusion. Remember the rule that parentheses in Perl may always be omitted—except when doing so would change the meaning of a statement. So, here are two ways to print the same thing:

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

So far, so good. But another rule in Perl is that if the invocation of `print` *looks* like a function call, then it *is* a function call. It's a simple rule, but what does it mean for something to look like a function call?

In a function call, there's a function name immediately followed by parentheses around the function's arguments, like this:

```
print (2+3);
```

That looks like a function call, so it is a function call. It prints `5`, but it returns a value like any other function. The return value of `print` is a true or false value, indicating the success of the print. It nearly always succeeds, unless you get some I/O error, so the `$result` in the following statement will normally be `1`:

```
$result = print("hello world!\n");
```

But what if you use the result in some other way? Suppose you decide to multiply the return value by four:

```
print (2+3)*4;  # Oops!
```

When Perl sees this line of code, it prints `5`, just as you asked. Then it takes the return value from `print`, which is `1`, and multiplies that by `4`. It then throws away the product, wondering why you didn't tell it to do something else with it. And at this point, someone looking over your

shoulder says, “Hey, Perl can’t do math! That should have printed `20` rather than `5`!”

This is the problem with the optional parentheses; sometimes we humans forget where the parentheses really belong. When there are no parentheses, `print` is a list operator, printing all of the items in the following list; that’s generally what you’d expect. But when the first thing after `print` is a left parenthesis, `print` is a function call, and it will print only what’s found inside the parentheses. Since that line had parentheses, it’s the same to Perl as if you’d said this:

```
( print(2+3) ) * 4;  # Oops!
```

Fortunately, Perl itself can almost always help you with this, if you ask for warnings—so use `-w`, or `use warnings`, at least during program development and debugging. To fix this, use more parentheses:

```
print( (2+3) * 4 );
```

Actually, this rule—“If it looks like a function call, it is a function call”—applies to all list functions in Perl, not just to `print`. It’s just that you’re most likely to notice it with `print`. If `print` (or another function name) is followed by an opening parenthesis, make sure that the corresponding closing parenthesis comes after *all* of the arguments to that function.

Formatted Output with printf

You may wish to have a little more control with your output than `print` provides. In fact, you may be accustomed to the formatted output of C’s `printf` function. Fear not! Perl provides a comparable operation with the same name.

The `printf` operator takes a template string followed by a list of things to print. That string is a fill-in-the-blanks template showing the desired form of the output:

```
printf "Hello, %s; your password expires in %d days!\n",
    $user, $days_to_die;
```

The template string holds a number of so-called *conversions*; each conversion begins with a percent sign (`%`) and ends with a letter. (As you'll see in a moment, there may be significant extra characters between these two symbols.) There should be the same number of items in the following list as there are conversions; if these don't match up, it won't work correctly. In the preceding example, there are two items and two conversions, so the output might look something like this:

```
Hello, merlyn; your password expires in 3 days!
```

There are many possible `printf` conversions, so we'll take time here to describe just the most common ones. Of course, the full details are available in the perlfunc documentation.

To print a number in what's generally a good way, use `%g` , which automatically chooses floating-point, integer, or even exponential notation, as needed:

```
printf "%g %g %g\n", 5/2, 51/17, 51 ** 17;  # 2.5 3 1.0683e+29
```

The `%d` format means a decimal integer, truncated as needed:

```
printf "in %d days!\n", 17.85;  # in 17 days!
```

Note that this is truncated, not rounded; you'll see how to round off a number in a moment.

There's also `%x` for hexadecimal and `%o` for octal if you need those:

```
printf "in 0x%x days!\n", 17;  # in Ox11 days!
printf "in 0%o days!\n", 17;  # in 021 days!
```

In Perl, you most often use `printf` for columnar data, since most formats accept a field width. If the data won't fit, the field will generally be expanded as needed:

```
printf "%6d\n", 42;  # output like ````42 (the ` symbol stands for a space)
printf "%2d\n", 2e3 + 1.95;  # 2001
```

The `%s` conversion means a string, so it effectively interpolates the given value as a string but with a given field width:

```
printf "%10s\n", "wilma";  # looks like `````wilma
```

A negative field width is left-justified (in any of these conversions):

```
printf "%-15s\n", "flintstone";  # looks like flintstone`````
```

The `%f` conversion (floating-point) rounds off its output as needed, and even lets you request a certain number of digits after the decimal point:

```
printf "%12f\n", 6 * 7 + 2/3;    # looks like ```42.666667
printf "%12.3f\n", 6 * 7 + 2/3;  # looks like ``````42.667
printf "%12.0f\n", 6 * 7 + 2/3;  # looks like ``````````43
```

To print a real percent sign, use `%%`, which is special in that it uses no element from the list:

```
printf "Monthly interest rate: %.2f%%\n",
    5.25/12;  # the value looks like "0.44%"
```

Maybe you thought you could simply put a backslash in front of the percent sign. Nice try, but no. The reason that won't work is that the format is an *expression*, and the expression `"\%"` means the one-character string `'%'`. Even if we got a backslash into the format string, `printf` wouldn't know what to do with it.

So far, you've specified the width of a field by putting it directly in the format string. You can also specify it as one of the arguments. A `*` inside the format string takes the next argument as a width:

```
printf "%*s", 10, "wilma";       # looks like `````wilma
```

You can use two `*` to get the total width and the number of decimal places to format a float:

```
printf "%*.*f", 6, 2, 3.1415926; # looks like ``3.14
printf "%*.*f", 6, 3, 3.1415926; # looks like `3.142
```

There's quite a bit more that you can do; see the `sprintf` documentation in perlfunc.

Arrays and printf

Generally, you won't use an array as an argument to `printf`. That's because an array may hold any number of items, and a given format string will work with only a certain fixed number of items.

But there's no reason you can't whip up a format string on the fly, since it may be any expression. This can be tricky to get right, though, so it may be handy (especially when debugging) to store the format in a variable:

```
my @items = qw( wilma dino pebbles );
my $format = "The items are:\n" . ("%10s\n" x @items);
## print "the format is >>$format<<\n"; # for debugging
printf $format, @items;
```

This uses the `x` operator (which you learned about in Chapter 2) to replicate the given string the number of times given by `@items` (which is being used in a scalar context). In this case, that's `3`, since there are three items, so the resulting format string is the same as if you wrote it as `"The items are:\n%10s\n%10s\n%10s\n"`. And the output prints each item on its own line, right-justified in a 10-character column, under a heading

line. Pretty cool, huh? But not cool enough, because you can even combine these:

```
printf "The items are:\n".("%10s\n" x @items), @items;
```

Note that here you have `@items` being used once in a scalar context, to get its length, and once in a list context, to get its contents. Context is important.

Filehandles

A *filehandle* is the name in a Perl program for an I/O connection between your Perl process and the outside world. That is, it's the name of a *connection*, not necessarily the name of a file. Indeed, Perl has mechanisms to connect a filehandle to almost anything.

Before Perl 5.6, all filehandle names were barewords, and Perl 5.6 added the ability to store a filehandle reference in a normal scalar variable. We'll show you the bareword versions first since Perl still uses those for its special filehandles, and catch up with the scalar variable versions later in this chapter.

You name these filehandles just like other Perl identifiers: letters, digits, and underscores (but not starting with a digit). The bareword filehandles don't have any prefix character, so Perl might confuse them with present or future reserved words, or with labels, which you'll see in Chapter 10. Once again, as with labels, the recommendation from Larry is that you use all uppercase letters in the name of your filehandle—not only does it stand out better, but it also guarantees that your program won't fail when Perl introduces a future (always lowercase) reserved word.

But there are also six special filehandle names that Perl already uses for its own purposes: `STDIN`, `STDOUT`, `STDERR`, `DATA`, `ARGV`, and `ARGVOUT`. Although you may choose any filehandle name you like, you shouldn't choose one of those six unless you intend to use that one's special properties.

Maybe you recognized some of those names already. When your program starts, `STDIN` is the filehandle naming the connection between the Perl process and wherever the program should get its input, known as the *standard input stream*. This is generally the user's keyboard unless the user asked for something else to be the source of input, such as a file or the output of another program through a pipe.

NOTE

The defaults for the three main I/O streams are what the Unix shells do by default. But it's not just shells that launch programs, of course. You'll see in Chapter 15 what happens when you launch another program from Perl.

There's also the *standard output stream*, which is `STDOUT`. By default, this one goes to the user's display screen, but the user may send the output to a file or to another program, as you'll see shortly. These standard streams come to you from the Unix "standard I/O" library, but they work in much the same way on most modern operating systems. The general idea is that your program should blindly read from `STDIN` and blindly write to `STDOUT`, trusting in the user (or generally whichever program is starting your program) to have set those up. In that way, the user can type a command like this one at the shell prompt:

```
$ ./your_program <dino >wilma
```

That command tells the shell that the program's input should be read from the file *dino*, and the output should go to the file *wilma*. As long as the program blindly reads its input from `STDIN`, processes it (in whatever way we need), and blindly writes its output to `STDOUT`, this will work just fine.

And at no extra charge, the program will work in a *pipeline*. This is another concept from Unix, which lets us write command lines like this one:

```
$ cat fred barney | sort | ./your_program | grep something | lpr
```

Now, if you're not familiar with these Unix commands, that's OK. This line says that the *cat* command should print out all of the lines of file *fred* followed by all of the lines of file *barney*. Then that output should be the input of the *sort* command, which sorts those lines and passes them on to *your_program*. After it has done its processing, *your_program* sends the data on to *grep*, which discards certain lines in the data, sending the others on to the *lpr* command, which should print everything that it gets on a printer. Whew!

Pipelines like that are common in Unix and many other systems today because they let you build powerful, complex commands out of simple, standard building blocks. Each building block does one thing very well, and it's your job to use them together in the right way.

There's one more standard I/O stream. If (in the previous example) *your_program* had to emit any warnings or other diagnostic messages, those shouldn't go down the pipeline. The *grep* command is set to discard anything that it hasn't specifically been told to look for, and so it will most likely discard the warnings. Even if it did keep the warnings, you probably don't want to pass them downstream to the other programs in the pipeline. So that's why there's also the *standard error stream*: `STDERR`. Even if the standard output is going to another program or file, the errors will go to wherever the user desires. By default, the errors will generally go to the user's display screen, but the user may send the errors to a file with a shell command like this one:

```
$ netstat | ./your_program 2>/tmp/my_errors
```

Generally, errors aren't buffered. That means that if the standard error and standard output streams are both going to the same place (such as the monitor), the errors may appear earlier than the normal output. For example, if your program prints a line of ordinary text, then tries to divide by zero, the output may show the message about dividing by zero first, and the ordinary text second.

Opening a Filehandle

So you've seen that Perl provides three filehandles— `STDIN`, `STDOUT`, and `STDERR`—which are automatically open to files or devices established by the program's parent process (probably the shell). When you need other filehandles, use the `open` operator to tell Perl to ask the operating system to open the connection between your program and the outside world. Here are some examples:

```
open CONFIG, 'dino';
open CONFIG, '<dino';
open BEDROCK, '>fred';
open LOG, '>>logfile';
```

The first one opens a filehandle called `CONFIG` to a file called *dino*. That is, the (existing) file *dino* will be opened and whatever it holds will come into our program through the filehandle named `CONFIG`. This is similar to the way that data from a file could come in through `STDIN` if the command line had a shell redirection like `<dino`. In fact, the second example uses exactly that sequence. The second does the same as the first, but the less-than sign explicitly says "use this filename for input," even though that's the default.

This may be important for security reasons. As you'll see in a moment (and in further detail in Chapter 15), there are a number of magical characters that may be used in filenames. If `$name` holds a user-chosen filename, simply opening `$name` will allow any of these magical characters to come into play. We recommend always using the three-argument form of `open`, which we'll show you in a moment.

Although you don't have to use the less-than sign to open a file for input, we include it because, as you can see in the third example, a greater-than sign means to create a new file for output. This opens the filehandle `BEDROCK` for output to the new file *fred*. Just as when the greater-than sign is used in shell redirection, we're sending the output to a *new* file called *fred*. If there's already a file of that name, you're asking to wipe it out and replace it with this new one.

The fourth example shows how you may use two greater-than signs (again, as the shell does) to open a file for appending. That is, if the file already exists, you will add new data at the end. If it doesn't exist, you will create it in much the same way as if you had used just one greater-than sign. This is handy for logfiles; your program could write a few lines to the end of a logfile each time it's run. So that's why the fourth example names the filehandle `LOG` and the file *logfile*.

You can use any scalar expression in place of the filename specifier, although typically you'll want to be explicit about the direction specification:

```
my $selected_output = 'my_output';
open LOG, "> $selected_output";
```

Note the space after the greater-than sign. Perl ignores this, but it keeps unexpected things from happening if `$selected_output` were `">passwd"`, for example (which would make an append instead of a write).

In modern versions of Perl (starting with Perl 5.6), you can use a "three-argument" `open`:

```
open CONFIG, '<', 'dino';
open BEDROCK, '>', $file_name;
open LOG, '>>', &logfile_name();
```

The advantage here is that Perl never confuses the mode (the second argument) with some part of the filename (the third argument), which has nice advantages for security. Since they are separate arguments, Perl doesn't have a chance to get confused.

The three-argument form has another big advantage. Along with the mode, you can specify an encoding. If you know that your input file is UTF-8, you can specify that by putting a colon after the file mode and naming the encoding:

```
open CONFIG, '<:encoding(UTF-8)', 'dino';
```

If you want to write your data to a file with a particular encoding, you do the same thing with one of the write modes:

```
open BEDROCK, '>:encoding(UTF-8)', $file_name;
open LOG, '>>:encoding(UTF-8)', &logfile_name();
```

There's a shortcut for this. Instead of the full `encoding(UTF-8)`, you might sometimes see `:utf8`. This isn't really a shortcut for the full version, because it doesn't care if the input (or output) is valid UTF-8. If you use `encoding(UTF-8)`, you ensure that the data is encoded correctly. The `:utf8` takes whatever it gets and marks it as a UTF-8 string even if it isn't, which might cause problems later. Still, you might see people do something like this:

```
open BEDROCK, '>:utf8', $file_name;  # probably not right
```

With the `encoding()` form, you can specify other encodings too. You can get a list of all of the encodings that your *perl* understands with a Perl one-liner:

```
$ perl -MEncode -le "print for Encode->encodings(':all')"
```

You should be able to use any of the names from that list as an encoding for reading or writing a file. Not all encodings are available on every machine, since the list depends on what you've installed (or excluded).

If you want a little-endian version of UTF-16:

```
open BEDROCK, '>:encoding(UTF-16LE)', $file_name;
```

Or perhaps Latin-1:

```
open BEDROCK, '>:encoding(iso-8859-1)', $file_name;
```

There are other *layers* that perform transformations on the input or output. For instance, you sometimes need to handle files that have DOS line endings, where each line ends with a carriage-return/linefeed (CR-LF) pair (also normally written as `"\r\n"`). Unix line endings only use the newlines. When you try to use one on the other, odd things can happen. The `:crlf` encoding takes care of that. When you want to ensure that you get a CR-LF at the end of each line, you can set that encoding on the file:

```
open BEDROCK, '>:crlf', $file_name;
```

Now when you print to each line, this layer translates each newline to a CR-LF, although be careful, since if you already have a CR-LF, you'll end up with two carriage returns in a row.

You can do the same thing to read a file, which might have DOS line endings:

```
open BEDROCK, '<:crlf', $file_name;
```

Now when you read a file, Perl will translate all CR-LFs to just newlines.

Binmoding Filehandles

You don't have to know the encoding ahead of time, or even specify it if you already know it. In older Perls, if you didn't want to translate line endings, such as a random value in a binary file that happens to have the same ordinal value as the newline, you used `binmode` to turn off line-ending processing:

```
binmode STDOUT; # don't translate line endings
binmode STDERR; # don't translate line endings
```

Perl 5.6 called it a *discipline*, but that name changed in favor of *layer*.

Starting with Perl 5.6, you could specify a layer as the second argument to `binmode`. If you want to output Unicode to `STDOUT`, you want to ensure that `STDOUT` knows how to handle what it gets:

```
binmode STDOUT, ':encoding(UTF-8)';
```

If you don't do that, you might get a warning (even without turning on warnings) because `STDOUT` doesn't know how you'd like to encode it:

```
Wide character in print at test line 1.
```

You can use `binmode` with either input or output handles. If you expect UTF-8 on standard input, you can tell Perl to expect that:

```
binmode STDIN, ':encoding(UTF-8)';
```

Bad Filehandles

Perl can't actually open a file all by itself. Like any other programming language, Perl can merely ask the operating system to let us open a file. Of course, the operating system may refuse because of permission settings, an incorrect filename, or other reasons.

If you try to read from a bad filehandle (that is, a filehandle that isn't properly open or a closed network connection), you'll see an immediate end-of-file. (With the I/O methods you'll see in this chapter, end-of-file will be indicated by `undef` in a scalar context or an empty list in a list context.) If you try to write to a bad filehandle, the data is silently discarded.

Fortunately, these dire consequences are easy to avoid. First of all, if you ask for warnings with `-w` or the `warnings` pragma, Perl will generally be able to tell you with a warning when it sees that you're using a bad filehandle. But even before that, `open` always tells you if it succeeded or failed by returning true for success or false for failure. So you could write code like this:

```
my $success = open LOG, '>>', 'logfile';  # capture the return value
if ( ! $success ) {
    # The open failed
    ...
}
```

Well, you *could* do it like that, but there's another way that you'll see in the next section.

Closing a Filehandle

When you are finished with a filehandle, you may close it with the `close` operator, like this:

```
close BEDROCK;
```

Closing a filehandle tells *perl* to inform the operating system that you're done with the given data stream, so it should write any last output data to disk in case someone is waiting for it. Perl automatically closes a filehandle if you reopen it (that is, if you reuse the filehandle name in a new `open`) or if you exit the program.

When it closes a filehandle, *perl* will flush any output buffers and release any locks on the file. Since someone else may be waiting for those things, a long-running program should generally close each filehandle as soon as possible. But many of our programs will take only one or two seconds to run to completion, so this may not matter. Closing a filehandle also releases possibly limited resources, so it's more than just being tidy.

Because of this, many simple Perl programs don't bother with `close`. But it's there if you want to be tidy, with one `close` for every `open`. In general, it's best to close each filehandle soon after you're done with it, though the end of the program often arrives soon enough.

Fatal Errors with die

Step aside for a moment. You need some stuff that isn't directly related to (or limited to) I/O but is more about getting out of a program earlier than normal.

When a fatal error happens inside Perl (for example, if you divide by zero, use an invalid regular expression, or call a subroutine that you haven't declared), your program stops with an error message telling why. But this functionality is available to you with the `die` function, so you can make your own fatal errors.

The `die` function prints out the message you give it (to the standard error stream, where such messages should go) and makes sure that your program exits with a nonzero exit status.

You may not have known it, but every program that runs in Unix (and many other modern operating systems) has an exit status, telling whether it was successful or not. Programs that run other programs (like the *make* utility program) look at that exit status to see that everything happened correctly. The exit status is just a single byte, so it can't say much; traditionally, it is 0 for success and a nonzero value for failure. Perhaps 1 means a syntax error in the command arguments, while 2 means that something went wrong during processing, and 3 means the configuration file couldn't be found; the details differ from one command to the next. But 0 always means that everything worked. When the exit status shows failure, a program like *make* knows not to go on to the next step.

So you could rewrite the previous example, perhaps as something like this:

```
if ( ! open LOG, '>>', 'logfile' ) {
  die "Cannot create logfile: $!";
}
```

If the `open` fails, `die` terminates the program and tells you that it cannot create the logfile. But what's that `$!` in the message? That's the human-readable complaint from the system. In general, when the system refuses to do something you've requested (such as opening a file), `$!` will give

you a reason (perhaps “permission denied” or “file not found,” in this case). This is the string that you may have obtained with `perror` in C or a similar language. This human-readable complaint message is available in Perl’s special variable `$!`.

It’s a good idea to include `$!` in the message when it could help the user figure out what they did wrong. But if you use `die` to indicate an error that is not the failure of a system request, don’t include `$!`, since it will generally hold an unrelated message left over from something Perl did internally. It will hold a useful value only immediately after a *failed* system request. A successful request won’t leave anything useful there.

There’s one more thing that `die` will do for you: it will automatically append the Perl program name and line number:

```
Cannot create logfile: permission denied at your_program line 1234.
```

That’s pretty helpful—in fact, you always seem to want more information in your error messages than you included the first time around. If you don’t want the line number and file revealed, make sure the dying words have a newline on the end. That is, another way you could use `die` is with a trailing newline on the message:

```
if (@ARGV < 2) {
  die "Not enough arguments\n";
}
```

If there aren’t at least two command-line arguments, that program will say so and quit. It won’t include the program name and line number, since the line number is of no use to the user; this is the user’s error, after all. As a rule of thumb, put the newline on messages that indicate a usage error and leave it off when the error might be something you want to track down during debugging.

You should always check the return value of `open`, since the rest of the program is relying on its success.

Warning Messages with warn

Just as `die` can indicate a fatal error that acts like one of Perl's built-in errors (like dividing by zero), you can use the `warn` function to cause a warning that acts like one of Perl's built-in warnings (like using an `undef` value as if it were defined, when warnings are enabled).

The `warn` function works just like `die` does, except for that last step—it doesn't actually quit the program. But it adds the program name and line number if needed, and it prints the message to standard error, just as `die` would.

Automatically die-ing

Starting with v5.10, the `autodie` pragma is part of the Standard Library. So far in the examples, you checked the return value of `open` and handled the error yourself:

```
if ( ! open LOG, '>>', 'logfile' ) {
  die "Cannot create logfile: $!";
}
```

That can get a bit tedious if you have to do that every time you want to open a filehandle. Instead, you can use the `autodie` pragma once in your program and automatically get the `die` if your `open` fails:

```
use autodie;

open LOG, '>>', 'logfile';
```

This pragma works by recognizing which Perl built-ins are system calls, which might fail for reasons beyond your program's control. When one of those system calls fails, `autodie` magically invokes the `die` on your behalf. Its error message looks close to what you might choose yourself:

```
Can't open('>>', 'logfile'): No such file or directory at test line 3
```

And having talked about death and dire warnings, we now return you to your regularly scheduled I/O instructional material. Read on.

Using Filehandles

Once a filehandle is open for reading, you can read lines from it just like you can read from standard input with `STDIN`. So, for example, to read lines from the Unix password file:

```
if ( ! open PASSWD, "/etc/passwd") {
  die "How did you get logged in? ($!)";
}

while (<PASSWD>) {
  chomp;
  ...
}
```

In this example, the `die` message uses parentheses around `$!`. Those are merely parentheses around the message in the output. (Sometimes a punctuation mark is just a punctuation mark.) As you can see, what we've been calling the "line-input operator" is really made of two components; the angle brackets (the *real* line-input operator) are around an input filehandle.

You can use a filehandle open for writing or appending with `print` or `printf`, appearing immediately after the keyword but before the list of arguments:

```
print LOG "Captain's log, stardate 3.14159\n";  # output goes to LOG
printf STDERR "%d percent complete.\n", $done/$total * 100;
```

Did you notice that there's no comma between the filehandle and the items to be printed? This looks especially weird if you use parentheses. Either of these forms is correct:

```
printf (STDERR "%d percent complete.\n", $done/$total * 100);
printf STDERR ("%d percent complete.\n", $done/$total * 100);
```

Changing the Default Output Filehandle

By default, if you don't give a filehandle to `print` (or to `printf`, as everything we say here about one applies equally well to the other), the output will go to `STDOUT`. But that default may be changed with the `select` operator. Here we'll send some output lines to `BEDROCK`:

```
select BEDROCK;
print "I hope Mr. Slate doesn't find out about this.\n";
print "Wilma!\n";
```

Once you've selected a filehandle as the default for output, it stays that way. But it's generally a bad idea to confuse the rest of the program, so you should generally set it back to `STDOUT` when you're done. Also by default, the output to each filehandle is buffered. Setting the special `$|` variable to `1` will set the currently selected filehandle (that is, the one selected at the time that the variable is modified) to always flush the buffer after each output operation. So if you wanted to be sure that the logfile gets its entries at once, in case you might be reading the log to monitor progress of your long-running program, you could use something like this:

```
select LOG;
$| = 1;  # don't keep LOG entries sitting in the buffer
select STDOUT;
# ... time passes, babies learn to walk, tectonic plates shift, and then...
print LOG "This gets written to the LOG at once!\n";
```

Reopening a Standard Filehandle

We mentioned earlier that if you were to reopen a filehandle (that is, if you were to open a filehandle `FRED` when you've already got an open

filehandle named `FRED`), the old one would be closed for you automatically. And we said that you shouldn't reuse one of the six standard filehandle names unless you intended to get that one's special features. And we also said that the messages from `die` and `warn`, along with Perl's internally generated complaints, go automatically to `STDERR`. If you put those three pieces of information together, you now have an idea about how you could send error messages to a file rather than to your program's standard error stream:

```
# Send errors to my private error log
if ( ! open STDERR, ">>/home/barney/.error_log") {
  die "Can't open error log for append: $!";
}
```

After reopening `STDERR`, any error messages from Perl go into the new file. But what happens if the `die` is executed—where will *that* message go, if the new file couldn't be opened to accept the messages?

The answer is that if one of the three system filehandles— `STDIN`, `STDOUT`, or `STDERR` —fails to reopen, Perl kindly restores the original one. That is, Perl closes the original one (of those three) only when it sees that opening the new connection is successful. Thus, this technique could be used to redirect any (or all) of those three system filehandles from inside your program, almost as if the program had been run with that I/O redirection from the shell in the first place.

Output with say

Perl 5.10 borrowed the `say` built-in from the ongoing development of Raku (which may have borrowed its `say` from Pascal's `println`). It's the same as `print`, although it adds a newline to the end. These forms all output the same thing:

```
use v5.10;

print "Hello!\n";
```

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

To just print a variable's value followed by a newline, we don't need to create an extra string or `print` a list. We just `say` the variable. This is especially handy in the common case of simply wanting to put a newline after whatever we want to output:

```
use v5.10;

my $name = 'Fred';
print "$name\n";
print $name, "\n";
say $name;
```

To interpolate an array, we still need to quote it, though. It's the quoting that puts the spaces between the elements:

```
use v5.10;

my @array = qw( a b c d );
say @array;   # "abcd\n"
say "@array"; # "a b c d\n";
```

Just like with `print`, we can specify a filehandle with `say`:

```
use v5.10;

say BEDROCK "Hello!";
```

Since this is a Perl 5.10 feature, though, we'll only use it when we are otherwise using a Perl 5.10 feature. The old, trusty `print` is still as good as it ever was, but we suspect that there will be some Perl programmers out there who want the immediate savings of not typing the four extra characters (two in the name, plus the `\n`).

Filehandles in a Scalar

Since v5.6, you can create a filehandle in a scalar variable so that you don't have to use a bareword. This makes many things, such as passing filehandles as subroutine arguments, storing them in arrays or hashes, or controlling their scope, much easier. Although you still need to know how to use the barewords, because you'll still find them in Perl code and they are actually quite handy in short scripts where you don't benefit that much from the filehandles in a variable.

If you use a scalar variable without a value in place of the bareword in `open`, your filehandle ends up in the variable. People typically do this with a lexical variable since that ensures that you get a variable without a value; some like to put a `_fh` on the end of these variable names to remind themselves that they are using it for a filehandle:

```
my $rocks_fh;
open $rocks_fh, '<', 'rocks.txt'
  or die "Could not open rocks.txt: $!";
```

You can even combine those two statements so that you declare the lexical variable right in the `open`:

```
open my $rocks_fh, '<', 'rocks.txt'
  or die "Could not open rocks.txt: $!";
```

Once you have the filehandle in your scalar variable, you use the variable, sigil and all, in the same place that you used the bareword version:

```
while( <$rocks_fh> ) {
  chomp;
  ...
}
```

This works for output filehandles too. You open the filehandle with the appropriate mode, then use the scalar variable in place of the bareword

filehandle:

```
open my $rocks_fh, '>>', 'rocks.txt'
  or die "Could not open rocks.txt: $!";
foreach my $rock ( qw( slate lava granite ) ) {
  say $rocks_fh $rock
}

print $rocks_fh "limestone\n";
close $rocks_fh;
```

Notice that you still don't use a comma after the filehandle in these examples. Perl realizes that `$rocks_fh` is a filehandle because there's no comma after the first thing following `print`. If you put a comma after the filehandle, your output looks odd. This probably isn't what you want to do:

```
print $rocks_fh, "limestone\n"; # WRONG
```

That example produces something like this:

```
GLOB(0xABCDEF12)limestone
```

What happened? Since you used the comma after the first argument, Perl treated that first argument as a string to print instead of the filehandle. Although we don't talk about references until the next book, *Intermediate Perl*, you're seeing a *stringification* of the reference instead of using it as you probably intend. This also means that these two are subtly different:

```
print STDOUT;
print $rocks_fh;  # WRONG, probably
```

In the first case, Perl knows that `STDOUT` is a filehandle because it is a bareword. Since there are no other arguments, it uses `$_` by default. In the second one, Perl can't tell what `$rock_fh` will have until it actually runs the statement. Since it doesn't know that it's a filehandle ahead of

time, it always assumes that the `$rock_fh` has a value you want to output. To get around this, you can always surround anything that should be a filehandle in braces to make sure that Perl does the right thing, even if you are using a filehandle that you stored in an array or a hash:

```
print { $rocks[0] } "sandstone\n";
```

When you use the braces, you don't print `$_` by default. You have to supply it yourself explicitly:

```
print { $rocks_fh } $_;
```

Depending on the sort of programming that you actually do, you might go one way or the other, choosing between bareword and scalar variable filehandles. For short programs, such as in system administration, barewords don't pose much of a problem. For big application development, you probably want to use the lexical variables to control the scope of your open filehandles.

Exercises

See "Answers to Chapter 5 Exercises" for answers to these exercises:

1. [7] Write a program that acts like *cat* but reverses the order of the output lines. (Some systems have a utility like this named *tac*.) If you run yours as `./tac fred barney betty`, the output should be all of file *betty* from last line to first, then *barney*, and then *fred*, also from last line to first. (Be sure to use the `./` in your program's invocation if you call it *tac* so that you don't get the system's utility instead!)
2. [8] Write a program that asks the user to enter a list of strings on separate lines, printing each string in a right-justified, 20-character column. To be certain that the output is in the proper columns, print a "ruler line" of digits as well. (This is simply a debugging aid.) Make sure that you're not using a 19-character column by mistake! For ex-

ample, entering `hello, good-bye` should give output something like this:

```
1234567890123456789012345678901234567890123456789012345678901234567890
                    hello
                  good-bye
```

3. [8] Modify the previous program to let the user choose the column width so that entering `30, hello, good-bye` (on separate lines) would put the strings at the 30th column. (Hint: see "Interpolation of Scalar Variables into Strings", about controlling variable interpolation.) For extra credit, make the ruler line longer when the selected width is larger.