Chapter 10. More Control Structures

In this chapter, you'll see some alternative ways to write Perl code. For the most part, these techniques don't make the language more powerful, but they do make it easier or more convenient to get the job done. You don't have to use these techniques in your own code, but don't be tempted to skip this chapter—you're certain to see these control structures in other people's code sooner or later (in fact, you're absolutely certain to see these things in use by the time you finish reading this book).

The unless Control Structure

In an `if` control structure, the block of code is executed only when the conditional expression is true. If you want to execute a block of code only when the conditional is false, change `if` to `unless`:

```
unless ($fred =~ /\A[A-Z_]\w*\z/i) {
  print "The value of \$fred doesn't look like a Perl identifier name.\n";
}
```

Using `unless` says to run the block of code *unless* this condition is true. It's just like using an `if` test with the opposite condition. Another way to say it is that it's like having the `else` clause on its own. That is, whenever you see an `unless` that you don't understand, you can rewrite it (either in your head or in reality) as an `if` test:

```
if ($fred =~ /\A[A-Z_]\w*\z/i) {
  # Do nothing
} else {
  print "The value of \$fred doesn't look like a Perl identifier name.\n";
}
```

It's no more or less efficient, and it should compile to the same internal bytecodes. Or, another way to rewrite it would be to negate the condi-

tional expression by using the negation operator (`!`):

```
if ( ! ($fred =~ /\A[A-Z_]\w*\z/i) ) {
  print "The value of \$fred doesn't look like a Perl identifier name.\n";
}
```

Generally, you should pick the way of writing code that makes the most sense to you, since that will probably make the most sense to your maintenance programmer. If it makes the most sense to write `if` with a negation, do that. More often, however, you'll probably find it natural to use `unless` .

The else Clause with unless

You could even have an `else` clause with an `unless` . While this syntax is supported, it's potentially confusing:

```
unless ($mon =~ /\AFeb/) {
  print "This month has at least thirty days.\n";
} else {
  print "Do you see what's going on here?\n";
}
```

Some people may wish to use this, especially when the first clause is very short (perhaps only one line) and the second is several lines of code. But you could make this one a negated `if` , or maybe simply swap the clauses to make a normal `if` :

```
if ($mon =~ /\AFeb/) {
  print "Do you see what's going on here?\n";
} else {
  print "This month has at least thirty days.\n";
}
```

It's important to remember that you're always writing code for two readers: the computer that will run the code and the human being who has to

keep the code working. If the human can't understand what you've written, pretty soon the computer won't be doing the right thing either.

The until Control Structure

Sometimes you want to reverse the condition of a `while` loop. To do that, just use `until`:

```
until ($j > $i) {
  $j *= 2;
}
```

This loop runs until the conditional expression returns true. But it's really just a `while` loop in disguise, except that this one repeats as long as the conditional is false rather than true. The conditional expression is evaluated before the first iteration, so this is still a zero-or-more-times loop, just like the `while` loop. As with `if` and `unless`, you could rewrite any `until` loop to become a `while` loop by negating the condition. But generally, you'll find it simple and natural to use `until` from time to time.

Statement Modifiers

In order to have a more compact notation, a statement may be followed by a modifier that controls it. For example, the `if` modifier works in a way analogous to an `if` block:

```
print "$n is a negative number.\n" if $n < 0;
```

That gives exactly the same result as if you had used this code, except that you saved some typing by leaving out the parentheses and curly braces:

```
if ($n < 0) {
  print "$n is a negative number.\n";
}
```

As we've said, Perl folks generally like to avoid typing. And the shorter form reads like in English: print this message if `$n` is less than zero.

Notice that the conditional expression is still evaluated first, even though it's written at the end. This is backward from the usual left-to-right ordering; in understanding Perl code, you have to do as Perl's internal compiler does, and read to the end of the statement before you can tell what it's really doing.

There are other modifiers as well:

```
&error("Invalid input") unless &valid($input);
$i *= 2 until $i > $j;
print " ", ($n += 2) while $n < 10;
&greet($_) foreach @person;
```

These all work just as (we hope) you would expect. That is, each one could be rewritten in a similar way to rewriting the `if` modifier example earlier. Here is one:

```
while ($n < 10) {
  print " ", ($n += 2);
}
```

The expression in parentheses inside the `print` argument list is noteworthy because it adds two to `$n`, storing the result back in `$n`. Then it returns that new value, which will be printed.

These shorter forms read almost like a natural language: call the `&greet` subroutine for each `@person` in the list. Double `$i` until it's larger than `$j`. One of the common uses of these modifiers is in a statement like this one:

```
print "fred is '$fred', barney is '$barney'\n" if $I_am_curious;
```

By writing the code "in reverse" like this, you can put the important part of the statement at the beginning. The point of that statement is to moni-

tor some variables; the point is not to check whether you're curious. Of course, we made up the name `$I_am_curious` ; it's not a built-in Perl variable. Generally, folks who use this technique will either call their variable `$TRACING` or will use a constant declared with the `constant` pragma. Some people prefer to write the whole statement on one line, perhaps with some tab characters before the `if` , to move it over toward the right margin as you saw in the previous example, while others put the `if` modifier indented on a new line:

```
print "fred is '$fred', barney is '$barney'\n"
  if $I_am_curious;
```

Although you can rewrite any of these expressions with modifiers as a block (the "old-fashioned" way), the converse isn't necessarily true. Perl allows only a single expression on either side of the modifier. So you can't write something `if` something `while` something `until` something `unless` something `foreach` something, which would just be too confusing. And you can't put multiple statements on the left of the modifier. If you need more than just a simple expression on either side, just write the code the old-fashioned way, with the parentheses and curly braces.

As we mentioned in relation to the `if` modifier, the control expression (on the right) is always evaluated first, just as it would be in the old-fashioned form.

With the `foreach` modifier, there's no way to choose a different control variable—it's always `$_` . Usually, that's not a problem, but if you want to use a different variable, you'll need to rewrite it as a traditional `foreach` loop.

The Naked Block Control Structure

The so-called "naked" block is one without a keyword or condition. That is, suppose you start with a `while` loop, which looks something like this:

```
while (condition) {
  body;
  body;
  body;
}
```

Now, take away the `while` keyword and the conditional expression, and you'll have a naked block:

```
{
  body;
  body;
  body;
}
```

The naked block is like a `while` or `foreach` loop, except that it doesn't loop; it just executes the body of the loop once, and it's done. It's an un-loop!

Later, you'll see that there are other uses for the naked block, but one of its features is that it provides a scope for temporary lexical variables:

```
{
  print "Please enter a number: ";
  chomp(my $n = <STDIN>);
  my $root = sqrt $n;  # calculate the square root
  print "The square root of $n is $root.\n";
}
```

In this block, `$n` and `$root` are temporary variables scoped to the block. As a general guideline, all variables should be declared in the smallest scope available. If you need a variable for just a few lines of code, you can put those lines into a naked block and declare the variable inside that block. Of course, if you need the value of either `$n` or `$root` later, you would need to declare them in a larger scope.

You may have noticed the `sqrt` function in that code and wondered about it—yes, it's a function we haven't shown before. Perl has many

built-in functions that are beyond the scope of this book. When you're ready, check the perlfunc documentation to learn about more of them.

The elsif Clause

Every so often, you may need to check a number of conditional expressions, one after another, to see which one of them is true. This can be done with the `if` control structure's `elsif` clause, as in this example:

```
if ( ! defined $dino) {
  print "The value is undef.\n";
} elsif ($dino =~ /^-?\d+\.?$/) {
  print "The value is an integer.\n";
} elsif ($dino =~ /^-?\d*\.\d+$/) {
  print "The value is a _simple_ floating-point number.\n";
} elsif ($dino eq '') {
  print "The value is the empty string.\n";
} else {
  print "The value is the string '$dino'.\n";
}
```

Perl will test the conditional expressions one after another. When one succeeds, the corresponding block of code is executed, and then the whole control structure is done and execution goes on to the rest of the program. If none has succeeded, the `else` block at the end is executed. (Of course, the `else` clause is still optional, although in this case it's often a good idea to include it.)

There's no limit to the number of `elsif` clauses, but remember that Perl has to evaluate the first 99 tests before it can get to the 100th. If you'll have more than half a dozen `elsif` s, you should consider whether there's a more efficient way to write it.

You may have noticed by this point that the keyword is spelled `elsif` , with only one `e` . If you write it as `elseif` with a second `e` , Perl will tell you it is not the correct spelling. Why? Because Larry says so.

Autoincrement and Autodecrement

You'll often want a scalar variable to count up or down by one. Since these are frequent constructs, there are shortcuts for them, like nearly everything else we do frequently.

The autoincrement operator (`++`) adds one to a scalar variable, like the same operator in C and similar languages:

```
my $bedrock = 42;
$bedrock++;  # add one to $bedrock; it's now 43
```

Just like other ways of adding one to a variable, the scalar will be created if necessary:

```
my @people = qw{ fred barney fred wilma dino barney fred pebbles };
my %count;                      # new empty hash
$count{$_}++ foreach @people;  # creates new keys and values as needed
```

The first time through that `foreach` loop, `$count{$_}` is incremented. That's `$count{"fred"}` , which thus goes from `undef` (since it didn't previously exist in the hash) up to `1` . The next time through the loop, `$count{"barney"}` becomes `1` ; after that, `$count{"fred"}` becomes `2` . Each time through the loop, you increment one element in `%count` , and possibly create it as well. After that loop finishes, `$count{"fred"}` is `3` . This provides a quick and easy way to see which items are in a list and how many times each one appears.

Similarly, the autodecrement operator (`--`) subtracts one from a scalar variable:

```
$bedrock--;  # subtract one from $bedrock; it's 42 again
```

The Value of Autoincrement

You can fetch the value of a variable and change that value at the same time. Put the `++` operator in front of the variable name to increment the variable first and then fetch its value. This is a *pre-increment*:

```
my $m = 5;
my $n = ++$m;  # increment $m to 6, and put that value into $n
```

Or put the `--` operator in front to decrement the variable first and then fetch its value. This is a *pre-decrement*:

```
my $c = --$m;  # decrement $m to 5, and put that value into $c
```

Here's the tricky part. Put the variable name first to fetch the value first, then do the increment or decrement. This is called a *post-increment* or *post-decrement*:

```
my $d = $m++;  # $d gets the old value (5), then increment $m to 6
my $e = $m--;  # $e gets the old value (6), then decrement $m to 5
```

It's tricky because you're doing two things at once. You're fetching the value, and you're changing it in the same expression. If the operator is first, you increment (or decrement) first, then use the new value. If the variable is first, you return its (old) value first, then do the increment or decrement. Another way to say it is that these operators return a value, but they also have the side effect of modifying the variable's value.

If you write these in an expression on their own, not using the value but only the side effect, there's no difference whether you put the operator before or after the variable:

```
$bedrock++;  # adds one to $bedrock
++$bedrock;  # just the same; adds one to $bedrock
```

A common use of these operators is in connection with a hash, to identify an item you have seen before:

```
my @people = qw{ fred barney bamm-bamm wilma dino barney betty pebbles };
my %seen;

foreach (@people) {
  print "I've seen you somewhere before, $_!\n"
    if $seen{$_}++;
}
```

When `barney` shows up for the first time, the value of `$seen{$_}++` is false, since it's the value of `$seen{$_}`, which is `$seen{"barney"}`, which is `undef`. But that expression has the side effect of incrementing `$seen{"barney"}`. When `barney` shows up again, `$seen{"barney"}` is now a true value, so you print the message.

The for Control Structure

Perl's `for` control structure is like the common `for` control structure you may have seen in other languages such as C. It looks like this:

```
for (initialization; test; increment) {
  body;
  body;
}
```

To Perl, though, this kind of loop is really a `while` loop in disguise, something like this:

```
initialization;
while (test) {
  body;
  body;
  increment;
}
```

The most common use of the `for` loop, by far, is for making computed iterations:

```
for ($i = 1; $i <= 10; $i++) {  # count from 1 to 10
  print "I can count to $i!\n";
}
```

When you've seen these before, you'll know what the first line is saying even before you read the comment. Before the loop starts, the control variable, `$i`, is set to `1`. Then the loop is really a `while` loop in disguise, looping while `$i` is less than or equal to `10`. Between each iteration and the next is the increment, which here is a literal increment, adding one to the control variable, which is `$i`.

The first time through this loop, `$i` is `1`. Since that's less than or equal to `10`, you see the message. Although the increment is written at the top of the loop, it logically happens at the bottom of the loop, after printing the message. So, `$i` becomes `2`, which is less than or equal to `10`, so we print the message again, and `$i` is incremented to `3`, which is less than or equal to `10`, and so on.

Eventually, you print the message that your program can count to `9`. Then you increment `$i` to `10`, which is less than or *equal* to `10`, so you run the loop one last time and print that your program can count to `10`. Finally, you increment `$i` for the last time, to `11`, which is not less than or equal to `10`. So control drops out of the loop, and you're on to the rest of the program.

All three parts are together at the top of the loop so that it's easy for an experienced programmer to read that first line and say, "Ah, it's a loop that counts `$i` from 1 to 10."

Note that after the loop finishes, the control variable has a value "after" the loop. That is, in this case, the control variable has gone all the way to `11`. This loop is very versatile, since you can make it count in all sorts of ways. For example, you can count down from 10 to 1:

```
for ($i = 10; $i >= 1; $i--) {
  print "I can count down to $i\n";
}
```

And this loop counts from `-150` up to `1000` by threes:

```
for ($i = -150; $i <= 1000; $i += 3) {
  print "$i\n";
}
```

It never gets to `1000` exactly. The last iteration uses `999`, since each value of `$i` is a multiple of three.

In fact, you could make any of the three control parts (initialization, test, or increment) empty if you wish, but you still need the two semicolons. In this (quite unusual) example, the test is a substitution, and the increment is empty:

```
for ($_ = "bedrock"; s/(.)//; ) {  # loops while the s/// is successful
  print "One character is: $1\n";
}
```

The test expression (in the implied `while` loop) is the substitution, which returns a true value if it succeeded. In this case, the first time through the loop, the substitution removes the `b` from `bedrock`. Each iteration removes another letter. When the string is empty, the substitution will fail, and the loop is done.

If the test expression (the one between the two semicolons) is empty, it's automatically true, making an infinite loop. But don't make an infinite loop like this until you see how to break out of such a loop, which we'll show later in this chapter:

```
for (;;) {
  print "It's an infinite loop!\n";
}
```

A more Perl-like way to write an intentional infinite loop, when you really want one, is with `while`:

```
while (1) {
  print "It's another infinite loop!\n";
}
```

If you somehow made an infinite loop that's gotten away from you, try Ctrl-C to halt your program.

Although C programmers are familiar with the first way, even a beginning Perl programmer should recognize that `1` is always true, making an intentional infinite loop, so the second is generally a better way to write it. Perl is smart enough to recognize a constant expression like that and optimize it away, so there's no difference in efficiency.

The Secret Connection Between foreach and for

It turns out that, inside the Perl parser, the keyword `foreach` is exactly equivalent to the keyword `for`. That is, any time Perl sees one of them, it's the same as if you had typed the other. Perl can tell which you meant by looking inside the parentheses. If you've got the two semicolons, it's a computed `for` loop (like we've just been talking about). If you don't have the semicolons, it's really a `foreach` loop:

```
for (1..10) {  # really a foreach loop from 1 to 10
  print "I can count to $_!\n";
}
```

That's really a `foreach` loop, but it's written `for`. Perl figures it out based on what it finds in the parentheses. If it finds the semicolons, it's the C-style `for`. Except for this one example, throughout this book we'll spell out `foreach` wherever it appears. How you do it is an issue of your personal style.

In Perl, the true `foreach` loop is almost always a better choice. In the `foreach` loop (written `for`) in that previous example, it's easy to see at a glance that the loop will go from `1` to `10`. But do you see what's wrong with this computed loop that's trying to do the same thing?

```
for ($i = 1; $i < 10; $i++) {  # Oops! Something is wrong here!
  print "I can count to $i!\n";
}
```

You're going to make this error, probably for the rest of your life. Do you see it yet? You have the right numbers in the statement but your comparison is off. Since 10 is not less than 10, this version actually counts up to 9. This is an *off by one* error. You can fix that with a single character:

```
for ($i = 1; $i <= 10; $i++) {  # OK now
  print "I can count to $i!\n";
}
```

Loop Controls

As you've surely noticed by now, Perl is one of the so-called "structured" programming languages. In particular, there's just one entrance to any block of code, which is at the top of that block. But there are times when you may need more control or versatility than what we've shown so far. For example, you may need to make a loop like a `while` loop, but one that always runs at least once. Or maybe you need to occasionally exit a block of code early. Perl has three loop control operators you can use in loop blocks to make the loop do all sorts of tricks.

The last Operator

The `last` operator immediately ends execution of the loop. (If you've used the "break" operator in C or a similar language, it's like that.) It's the "emergency exit" for loop blocks. When you hit `last`, the loop is done.

For example:

```
# Print all input lines mentioning fred, until the __END__ marker
while (<STDIN>) {
  if (/__END__/) {
    # No more input on or after this marker line
```

```
    last;
  } elsif (/fred/) {
    print;
  }
}
## last comes here ##
```

Once an input line has the `__END__` marker, that loop is done. Of course, that comment line at the end is merely a comment—it's not required in any way. We just threw that in to make it clearer what's happening.

There are five kinds of loop blocks in Perl. These are the blocks of `for`, `foreach`, `while`, `until`, and the naked block. The curly braces of an `if` block or subroutine don't qualify. As you may have noticed in this example, the `last` operator applied to the entire loop block.

The `last` operator will apply to the innermost currently running loop block. To jump out of outer blocks, stay tuned; that's coming up in a little bit.

The next Operator

Sometimes you're not ready for the loop to finish, but you're done with the current iteration. That's what the `next` operator is good for. It jumps to the *inside* of the bottom of the current loop block. After `next`, control continues with the next iteration of the loop (much like the `continue` operator in C or a similar language):

```
# Analyze words in the input file or files
while (<>) {
  foreach (split) {  # break $_ into words, assign each to $_ in turn
    $total++;
    next if /\W/;    # strange words skip the remainder of the loop
    $valid++;
    $count{$_}++;    # count each separate word
    ## next comes here ##
  }
}
```

```
print "total things = $total, valid words = $valid\n";
foreach $word (sort keys %count) {
  print "$word was seen $count{$word} times.\n";
}
```

This one is a little more complex than most of our examples up to this point, so let's take it step by step. The `while` loop is reading lines of input from the diamond operator, one after another, into `$_`; you've seen that before. Each time through that loop, another line of input will be in `$_`.

Inside that loop, the `foreach` loop iterates over the return value of `split`. Do you remember the default for `split` with no arguments? That splits `$_` on whitespace, in effect breaking `$_` into a list of words. Since the `foreach` loop doesn't mention some other control variable, the control variable will be `$_`. So, you'll see one word after another in `$_`.

But didn't we just say that `$_` holds one line of input after another? Well, in the outer loop, that's what it holds. But inside the `foreach` loop, it holds one word after another. It's not a problem for Perl to reuse `$_` for a new purpose; this happens all the time.

Now, inside the `foreach` loop, you're seeing one word at a time in `$_`. `$total` is incremented, so it must be the total number of words. But the next line (which is the point of this example) checks to see whether the word has any nonword characters—anything but letters, digits, and underscores. So, if the word is `Tom's`, or if it is `full-sized`, or if it has an adjoining comma, quote mark, or any other strange character, it will match that pattern and you'll skip the rest of the loop, going on to the next word.

But let's say that it's an ordinary word, like `fred`. In that case, you count `$valid` up by one, and also `$count{$_}`, keeping a count for each different word. So, when you finish the two loops, you've counted every word in every line of input from every file the user wanted you to use.

We're not going to explain the last few lines. By now, we hope you've got stuff like that down already.

Like `last`, `next` may be used in any of the five kinds of loop blocks: `for`, `foreach`, `while`, `until`, or the naked block. Also, if you nest loop blocks, `next` works with the innermost one. You'll see how to change that at the end of this section.

The redo Operator

The third member of the loop control triad is `redo`. It says to go back to the top of the current loop block, without testing any conditional expression or advancing to the next iteration. (If you've used C or a similar language, you've never seen this one before. Those languages don't have this kind of operator.) Here's an example:

```
# Typing test
my @words = qw{ fred barney pebbles dino wilma betty };
my $errors = 0;

foreach (@words) {
  ## redo comes here ##
  print "Type the word '$_': ";
  chomp(my $try = <STDIN>);
  if ($try ne $_) {
    print "Sorry - That's not right.\n\n";
    $errors++;
    redo;  # jump back up to the top of the loop
  }
}
print "You've completed the test, with $errors errors.\n";
```

Like the other two operators, `redo` will work with any of the five kinds of loop blocks, and it will work with the innermost loop block when they're nested.

The big difference between `next` and `redo` is that `next` will advance to the next iteration, but `redo` will redo the current iteration. Here's an example program that you can play with to get a feel for how these three operators work:

```
foreach (1..10) {
  print "Iteration number $_.\n\n";
  print "Please choose: last, next, redo, or none of the above? ";
  chomp(my $choice = <STDIN>);
  print "\n";
  last if $choice =~ /last/i;
  next if $choice =~ /next/i;
  redo if $choice =~ /redo/i;
  print "That wasn't any of the choices... onward!\n\n";
}

print "That's all, folks!\n";
```

If you just press Return without typing anything (try it two or three times), the loop counts along from one number to the next. If you choose `last` when you get to number four, the loop is done, and you won't go on to number five. If you choose `next` when you're on four, you're on to number five without printing the "onward" message. And if you choose `redo` when you're on four, you're back to doing number four all over again.

Labeled Blocks

When you need to work with a loop block that's not the innermost one, use a label. Labels in Perl are like other identifiers—made of letters, digits, and underscores, but they can't start with a digit. However, since they have no prefix character, labels could be confused with the names of built-in function names, or even with your own subroutines' names. So it would be a poor choice to make a label called `print` or `if`. Because of that, Larry recommends that they be all uppercase. That not only ensures that the label won't conflict with another identifier but it also makes it easy to spot the label in the code. In any case, labels are rare, only showing up in a small percentage of Perl programs.

To label a loop block, just put the label and a colon in front of the loop. Then, inside the loop, you may use the label after `last`, `next`, or `redo`, as needed:

```
LINE: while (<>) {
  foreach (split) {
    last LINE if /__END__/;  # bail out of the LINE loop
    ...
  }
}
```

For readability, it's generally nice to put the label at the left margin, even if the current code is at a higher indentation. Notice that the label names the entire block; it's not marking a target point in the code. In that previous snippet of sample code, the special `__END__` token marks the end of all input. Once that token shows up, the program will ignore any remaining lines (even from other files).

It often makes sense to choose a noun as the name of the loop. That is, the outer loop is processing a line at a time, so we called it `LINE`. If we had to name the inner loop, we would have called it `WORD`, since it processes a word at a time. That makes it convenient to say things like "(move on to the) `next WORD`" or "`redo` (the current) `LINE`":

```
LINE: while (<>) {
  WORD: foreach (split) {
    last LINE if /__END__/;  # bail out of the LINE loop
    last WORD if /EOL/;      # skip the rest of the line
    ...
  }
}
```

The Conditional Operator

When Larry was deciding which operators to make available in Perl, he didn't want former C programmers to miss something that C had and Perl didn't, so he brought over all of C's operators to Perl. That meant bringing over C's most confusing operator: the conditional operator `?:`. While it may be confusing, it can also be quite useful.

The “conditional operator” is like an if-then-else test, all rolled into an expression. It’s sometimes called a “ternary operator” because it takes three operands. It looks like this:

```
expression ? if_true_expr : if_false_expr
```

NOTE

Some people call the “conditional operator” the “ternary operator.” It does take three parts, and that’s enough to distinguish it from the other Perl operators. Older Perlers will still say “ternary,” but it’s not a good habit to develop if you don’t already have it.

First, Perl evaluates the expression to see whether it’s true or false. If it’s true, Perl returns the second expression; otherwise, it returns the third expression. Every time, one of the two expressions on the right is evaluated, and one is ignored. That is, if the first expression is true, then the second expression is evaluated, and the third is ignored. If the first expression is false, then the second is ignored, and the third is evaluated as the value of the whole thing.

In this example, the result of the subroutine `&is_weekend` determines which string expression you’ll assign to the variable:

```
my $location = &is_weekend($day) ? "home" : "work";
```

And here, you calculate and print out an average—or just a placeholder line of hyphens, if there’s no average available:

```
my $average = $n ? ($total/$n) : "-----";
print "Average: $average\n";
```

You could always rewrite any use of the `?:` operator as an `if` structure, often much less conveniently and less concisely:

```
my $average;
if ($n) {
  $average = $total / $n;
} else {
  $average = "-----";
}
print "Average: $average\n";
```

Here's a trick you might see used to code up a nice multiway branch:

```
my $size =
  ($width < 10) ? "small"  :
  ($width < 20) ? "medium" :
  ($width < 50) ? "large"  :
          "extra-large"; # default
```

That is really just three nested `?:` operators, and it works quite well once you get the hang of it.

Of course, you're not obliged to use this operator. Beginners may wish to avoid it. But you'll see it in others' code sooner or later, and we hope that one day you'll find a good reason to use it in your own programs.

Logical Operators

As you might expect, Perl has all of the necessary logical operators needed to work with Boolean (true/false) values. For example, it's often useful to combine logical tests by using the logical AND operator (`&&`) and the logical OR operator (`||`):

```
if ($dessert{'cake'} && $dessert{'ice cream'}) {
  # Both are true
  print "Hooray! Cake and ice cream!\n";
} elsif ($dessert{'cake'} || $dessert{'ice cream'}) {
  # At least one is true
  print "That's still good...\n";
} else {
```

```
  # Neither is true; do nothing (we're sad)
}
```

There may be a shortcut. If the left side of a logical AND operation is false, the whole thing is false, since logical AND needs both sides to be true in order to return true. In that case, there's no reason to check the right side, so Perl doesn't evaluate it. Consider what happens in this example if `$hour` is `3`:

```
if ( (9 <= $hour) && ($hour < 17) ) {
  print "Aren't you supposed to be at work...?\n";
}
```

Similarly, if the left side of a logical OR operation is true, Perl doesn't evaluate the right side. Consider what happens here if `$name` is `fred`:

```
if ( ($name eq 'fred') || ($name eq 'barney') ) {
  print "You're my kind of guy!\n";
}
```

Because of this behavior, these operators are called "short-circuit" logical operators. They take a short circuit to the result whenever they can. In fact, it's fairly common to rely on this short-circuit behavior. Suppose you need to calculate an average:

```
if ( ($n != 0) && ($total/$n < 5) ) {
  print "The average is below five.\n";
}
```

In that example, Perl evaluates the right side only if the left side is true so that you can't accidentally divide by zero and crash the program (and we'll show you more about that in "Trapping Errors").

The Value of a Short-Circuit Operator

Unlike what happens in C (and similar languages), the value of a short-circuit logical operator is the last part evaluated, not just a Boolean value.

This provides the same result, in that the last part evaluated is always true when the whole thing should be true, and it's always false when the whole thing should be false.

But it's a much more useful return value. Among other things, the logical OR operator is quite handy for selecting a default value:

```
my $last_name = $last_name{$someone} || '(No last name)';
```

If `$someone` is not listed in the hash, the left side will be `undef`, which is false. So the logical OR will have to look to the right side for the value, making the right side the default. In this idiom, the default value won't merely replace `undef`; it would replace any false value equally well. You could fix that with the conditional operator:

```
my $last_name = defined $last_name{$someone} ?
  $last_name{$someone} : '(No last name)';
```

That's too much work, and you had to say `$last_name{$someone}` twice. Perl 5.10 added a better way to do this, and it's discussed in the next section.

The defined-or Operator

In the previous section, you used the `||` operator to give a default value. That ignored the special case where the defined value was false but perfectly acceptable as a value. You then saw the uglier version using the conditional operator.

Perl 5.10 got around this sort of bug with the defined-or operator, `//`, which short-circuits when it finds a defined value, no matter if that value on the lefthand side is true or false. Even if someone's last name is `0`, this version still works because it won't replace defined values:

```
use v5.10;
```

```
my $last_name = $last_name{$someone} // '(No last name)';
```

Sometimes you just want to give a variable a value if it doesn't already have one, and if it already has a value, to leave it alone. Suppose you want to only print messages if you set the `VERBOSE` environment variable. You check the value for the `VERBOSE` key in the `%ENV` hash. If it doesn't have a value, you want to give it one:

```
use v5.10;

my $Verbose = $ENV{VERBOSE} // 0; # off by default
print "I can talk to you!\n" if $Verbose;
```

You can see this in action by trying several values with `//` to see which ones pass through to the `default` value:

```
use v5.10;

foreach my $try ( 0, undef, '0', 1, 25 ) {
  print "Trying [$try] ---> ";
  my $value = $try // 'default';
  say "\tgot [$value]";
}
```

The output shows that you only get the `default` string when `$try` is `undef`:

```
Trying [0] --->     got [0]
Trying [] --->      got [default]
Trying [0] --->     got [0]
Trying [1] --->     got [1]
Trying [25] --->    got [25]
```

Sometimes you want to set a value when there isn't one already. For instance, when you have warnings enabled and try to print an undefined value, you get an annoying error:

```
use warnings;

my $name;  # no value, so undefined!
printf "%s", $name; # Use of uninitialized value in printf ...
```

Sometimes that error is harmless. You could just ignore it, but if you expect that you might try to print an undefined value, you can use the empty string instead:

```
use v5.10;
use warnings;

my $name;  # no value, so undefined!
printf "%s", $name // '';
```

Control Structures Using Partial-Evaluation Operators

The four operators that you've just seen— `&&`, `||`, `//`, and `?:` —all share a peculiar property: depending on the value on the left side, they may or may not evaluate an expression. Sometimes they evaluate the expression and sometimes they don't. For that reason, these are sometimes called *partial-evaluation* operators, since they may not evaluate all of the expressions around them. And partial-evaluation operators are automatically control structures. It's not as if Larry felt a burning need to add more control structures to Perl. But once he had decided to put these partial-evaluation operators into Perl, they automatically became control structures as well. After all, anything that can activate and deactivate a chunk of code is, by that very fact, a control structure.

Fortunately, you'll notice this only when the controlled expression has side effects, like altering a variable's value or causing some output. For example, suppose you ran across this line of code:

```
($m < $n) && ($m = $n);
```

Right away, you should notice that the result of the logical AND isn't being assigned anywhere. Why not?

If `$m` is really less than `$n`, the left side is true, so the right side will be evaluated, thereby doing the assignment. But if `$m` is not less than `$n`, the left side will be false, and thus the right side would be skipped. So that line of code would do essentially the same thing as this one, which is easier to understand:

```
if ($m < $n) { $m = $n }
```

Or maybe even:

```
$m = $n if $m < $n;
```

Or maybe you're fixing someone else's program, and you see a line like this one:

```
($m > 10) || print "why is it not greater?\n";
```

If `$m` is really greater than 10, the left side is true and the logical OR is done. But if it's not, the left side is false, and this will go on to print the message. Once again, this could (and probably should) be written in the traditional way, probably with `if` or `unless`. You most often see this sort of expression from people coming from the shell scripting world and transferring the idioms they know there into their new language.

If you have a particularly twisted brain, you might even learn to read these lines as if they were written in English. For example: check that `$m` is less than `$n`, *and if it is*, then do the assignment. Check that `$m` is more than 10, *or if it's not*, then print the message.

It's generally former C programmers or old-time Perl programmers who most often use these ways of writing control structures. Why do they do it? Some have the mistaken idea that these are more efficient. Some think

these tricks make their code cooler. Some are merely copying what they saw someone else do.

In the same way, you can use the conditional operator for control. In this case, you want to assign `$x` to the smaller of two variables:

```
($m < $n) ? ($m = $x) : ($n = $x);
```

If `$m` is smaller, it gets `$x`. Otherwise, `$n` does.

There is another way to write the logical AND and logical OR operators. You may wish to write them out as words: `and` and `or`. These word operators have the same behaviors as the ones written with punctuation, but the words are down at the bottom of the precedence chart. Since the words don't "stick" so tightly to the nearby parts of the expression, they may need fewer parentheses:

```
$m < $n and $m = $n;  # but better written as the corresponding if
```

There are also the low-precedence `not` (like the logical-negation operator, `!`) and the rare `xor`.

Then again, you may need *more* parentheses. Precedence is a bugaboo. Be sure to use parentheses to say what you mean, unless you're sure of the precedence. Nevertheless, since the word forms are very low precedence, you can generally understand that they cut the expression into big pieces, doing everything on the left first, and then (if needed) everything on the right.

Despite the fact that using logical operators as control structures can be confusing, sometimes they're the accepted way to write code. The idiomatic way of opening a file in Perl looks like this:

```
open my $fh, '<', $filename
  or die "Can't open '$filename': $!";
```

By using the low-precedence short-circuit `or` operator, you tell Perl that it should "open this file…or die!" If the `open` succeeds, returning a true value, the `or` is complete. But if it fails, the false value causes the `or` to evaluate the part on the right, which will `die` with a message.

So, using these operators as control structures is part of idiomatic Perl—Perl as she is spoken. Used properly, they can make your code more powerful; otherwise, they can make your code unmaintainable. Don't overuse them.

Exercises

See "Answers to Chapter 10 Exercises" for answers to these exercises:

1. [25] Make a program that will repeatedly ask the user to guess a secret number from 1 to 100 until the user guesses the secret number. Your program should pick the number at random by using the magical formula `int(1 + rand 100)`. See what the perlfunc documentation says about `int` and `rand` if you're curious about these functions. When the user guesses wrong, the program should respond "Too high" or "Too low." If the user enters the word `quit` or `exit`, or if the user enters a blank line, the program should quit. Of course, if the user guesses correctly, the program should quit then as well!
2. [10] Modify the program from the previous exercise to print extra debugging information as it goes along, such as the secret number it chose. Make your change such that you can turn it off, but your program emits no warnings if you turn it off. If you are using Perl 5.10 or later, use the `//` operator. Otherwise, use the conditional operator.
3. [10] Modify the program from Exercise 3 in Chapter 6 (the environment lister) to print `(undefined value)` for environment variables without a value. You can set the new environment variables in the program. Ensure that your program reports the right thing for variables with a false value. If you are using Perl 5.10 or later, use the `//` operator. Otherwise, use the conditional operator.